

CA Identity Manager

Programming Guide for Provisioning

r12



This documentation and any related computer software help programs (hereinafter referred to as the "Documentation") is for the end user's informational purposes only and is subject to change or withdrawal by CA at any time.

This Documentation may not be copied, transferred, reproduced, disclosed, modified or duplicated, in whole or in part, without the prior written consent of CA. This Documentation is confidential and proprietary information of CA and protected by the copyright laws of the United States and international treaties.

Notwithstanding the foregoing, licensed users may print a reasonable number of copies of the Documentation for their own internal use, and may make one copy of the related software as reasonably required for back-up and disaster recovery purposes, provided that all CA copyright notices and legends are affixed to each reproduced copy. Only authorized employees, consultants, or agents of the user who are bound by the provisions of the license for the Product are permitted to have access to such copies.

The right to print copies of the Documentation and to make a copy of the related software is limited to the period during which the applicable license for the Product remains in full force and effect. Should the license terminate for any reason, it shall be the user's responsibility to certify in writing to CA that all copies and partial copies of the Documentation have been returned to CA or destroyed.

EXCEPT AS OTHERWISE STATED IN THE APPLICABLE LICENSE AGREEMENT, TO THE EXTENT PERMITTED BY APPLICABLE LAW, CA PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IN NO EVENT WILL CA BE LIABLE TO THE END USER OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF CA IS EXPRESSLY ADVISED OF SUCH LOSS OR DAMAGE.

The use of any product referenced in the Documentation is governed by the end user's applicable license agreement.

The manufacturer of this Documentation is CA.

Provided with "Restricted Rights." Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in FAR Sections 12.212, 52.227-14, and 52.227-19(c)(1) - (2) and DFARS Section 252.227-7014(b)(3), as applicable, or their successors.

All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

Copyright © 2008 CA. All rights reserved.

CA Product References

This document references the following CA products:

- CA Identity Manager
- CA SiteMinder® Web Access Manager
- CA Security Command Center (SCC)
- CA Audit
- eTrust® Directory, also known as CA Directory

Contact Technical Support

For online technical assistance and a complete list of locations, primary service hours, and telephone numbers, contact Technical Support at <http://ca.com/support>.

Contents

Chapter 1: Provisioning SDK Overview 15

How This Guide Is Organized	15
Development System Prerequisites	16
System Recommendations	17
Provisioning SDK Installation	17
Installation Directories	18
Provisioning Server Terminology	18
Provisioning Server Architecture	19
Identity Manager	21
Provisioning Server	21
Provisioning Server Clients	26
Managed Endpoints	27
Common Objects Server (COS) Plug-in	27
Provisioning SDK Components	28
GUI Framework	28
Server Framework	28
Superagent Framework	29

Chapter 2: Provisioning SDK Fundamentals 31

The LDAP Protocol	31
LDAP Operations	31
LDAP Directories	32
Objects in an LDAP Directory	32
Object Attributes	33
Distinguished Names	34
Parser Tables	35
Schema Definition	36
Creating a Parser Table	36
The SCHEMAGEN Utility	37
Directory Information Trees (DITs)	38
Provisioning Server DIT	38
Provisioning Directory DIT	39
Connector Server DIT	39
Identifying DITs	39
Defining DITs	40
Processing LDAP Requests	40
Code Portability	41

Factors Affecting Portability	41
Header Files	42
Conditional Compilation	42
Portable printf() Processing	42
Converting Program Exit Code	43

Chapter 3: Implementing a Connector **45**

Connector Overview	45
Connector Design Specification	46
Implement a Custom Connector	48
Define Objects in the Schema	49
Build and Implement the GUI Plug-in	50
Build and Implement the Connector	51
Configure the Server Plug-in	52

Chapter 4: The Sample Connector **55**

Environment Variables	55
Building the Sample SDK Connector	56
Compiling the Connector and Generating its Schema	56
Compiling SDK Samples on Solaris	60
Adding the SDK Connector to the Provisioning Directory	61
Registering the Sample Directory	63
Exploring and Correlating SDK Directory Accounts	64
SDK Connector Code	65
Provisioning SDK Directories	65
Refresh the Visual C++ Compilation Environment	66
Upgrade Existing Custom Connectors	67

Chapter 5: Predefined Object Classes **69**

Base Classes	69
Define a Subclass to a Base Class	69
Common Object Classes	70
Namespace Class	70
Directory Class	70
Account Class	71
Policy Class	71
Global User Class	71
Program Exit Class	72
Inclusion Class	73
Inclusion Objects	73

Predefined Inclusions	74
Required Inclusions	74
Optional Relationships	75
Inclusion Sample Code	77
 Chapter 6: Endpoint Acquisition	 79
How to Register the Endpoint	79
How to Explore the Endpoint	80
Dynamic Containers	84
How to Correlate the Endpoint	84
How to Update Users	86
 Chapter 7: Superagent Server Plug-in	 89
SASP Object Processing	89
SASP Characteristics	90
The C++ Connector Server	90
Connector Server Functions	91
Running Agent DLLs Using the VC++ Debugger	91
How to Build an Agent Plug-In	92
How to Initialize an Agent Plug-in	92
How to Declare an Object Class	94
How to Construct an Agent DIT	95
Directory Information Tree (DIT)	100
Naming Endpoints, Object Classes, Objects, and Attributes	100
Creating Required Objects	101
Defining Objects	102
 Chapter 8: Building Parser Tables	 105
Parser Table Building Process	105
Naming Parser Table	107
View the Contents of Binary PTT Files	107
NAMESPACE Statement	108
CLASS Statement	109
KEYWORD Statement	115
Search and Long-valued Attribute Issues	125
Declare an Attribute as Non-Searchable	125
Create a Wide Index for an Attribute	126

Chapter 9: Building a GUI Plug-In **127**

Build a Parser Table for the GUI Plug-In	127
Namespace Class	128
Directory Class	128
Tree Hierarchy	128
GUI Callout Module	128
SDK Sample Connector Code	130
GUI Objects Definition	139
Resource ID Ranges	140
Custom Menus	140
Property Sheets and Pages	141
Property Sheets and Property Pages	142
Create the C++ Code for a Property Sheet	142
Property Page Code	145
Code Examples	147
Building and Linking the GUI Plug-in	161
Adding Objects to the Provisioning Directory	161
Linking and Providing Help Files	161
Install and Test the GUI Plug-in	162

Chapter 10: Building an Agent Plug-In **163**

Required Files to be Included	163
Deriving a Class for Each Node	164
Derive a Class for Static DITs	164
Derive a Class for Dynamic DITs	165
Implement the Functions for an Object Class	165
Implement the Factory Functions	166
Initialize the Agent Plug-In	167
Enable Thread Support of Agent Plug-Ins	168
Enable Parallel Search in an Agent Plug-In	170
Build an Agent Plug-In	171
Sample Call to Initialize the Agent Plug-In	171
Test the Agent Plug-in DLL	174

Chapter 11: The Sample ODBC-SDK Connector **175**

How to Customize the Sample ODBC-SDK Connector	175
Analyze the Database Application	177
Schema Objects Definition	181
Build and Implement the GUI Plug-In	191
Build and Implement the Agent Plug-in	191

Update the Populate Command	197
Configure the Superagent Server Plug-In	197
Create the KMS Connector	198
OdbcWrapper Class	200
AttributeMap	201
How to Install LibODBC++ and Configure the OSN Connector	202
How to Build the Sample ODBC-SDK Connector	203
Download and Build the libOdbc++ Library	204
Compile the Sample ODBC-SDK Connector	205
Add the ODBC-SDK Connector Object to Manager	206
Create a Sample ODBC Database (Optional)	207
Create a New ODBC Data Source Name (Optional)	208
Register the Sample ODBC-SDK Endpoint	209
Explore and Correlate Accounts	209
ODBC-SDK Connector Code	210

Chapter 12: Common Program Exits **211**

Program Exit Architecture	211
Program Exit Hierarchy and Order	212
Common Program Exit Structure	213
Program Exit Input Argument	213
Program Exit Return Value	215
Common Exits DLL Interface	218
Common Exits SOAP Interface	220
eTExitType	220
Valid Values for eTExitType	221
Containment	226
Custom Function Program Exits	230
Obscured Returned Values	232
Sample Flow/Execution Diagram	233
Code Examples	233

Chapter 13: Universal Provisioning Program Exits **235**

UPC Program Exit Architecture	236
Entry Points in Non-Managed Mode	236
Entry Points in Managed Mode	236
UPC Program Exit Objects	237
Program Exit Order	237
Implementing UPC Program Exits in Multiple Domain Environments	237
UPC Program Exit Structure	238
Input XML Buffer (Input Argument)	238

XML Exit Custom Data Block	244
Return XML Buffer	245
UPC Exits DLL Interface	246
UPC Exits SOAP Interface.....	246
eTExitType	246
Valid Values for etExitType	246
Sample Flow/Execution Diagrams.....	249
Flowchart for Non-Managed Mode Exits.....	249
Flowchart for Managed Mode Exits	251
Code Examples	252
Non-Managed Mode Exit Examples	252
Managed Mode Exit Examples	253
Common Exit with UPC Example	253

Chapter 14: Supporting Program Exits In Connectors 255

Execution Flow (Logic)	255
Pre-Exits	255
Operation	256
Post-Exit	256
Support for Common Exits	257
Parser Table Enhancement	257
GUI Plug-In Enhancement	257
Agent Plug-In Enhancement	257
Support for Native Exits.....	257
Parser Table Enhancement	257
GUI Plug-In Enhancement	258
Agent Plug-in Enhancement	259
Exit Types.....	261
Exit Type Functionality	261
Code Examples for Program Exits in Options.....	262

Appendix A: Provisioning Schema and Structure 263

Provisioning Schema	263
COSSchemaAbridged.txt File	263
COSSchemaUnabridged.txt File	264
File Formats	264
Endpoint Structure.....	266
Distinguished Names	266
Common Object DITs	267
User-Friendly Object Names	270
Object Hierarchy	272

Appendix B: GUI Framework API Functions **273**

apply()	273
setData()	275
setErrorField()	277

Appendix C: GUI Callout Functions **279**

CanDuplicateEntry	279
CreateInclusion	280
DeleteEntry	282
DuplicateEntry	283
GetPropertySheetID	284
IsValidDrag	286
OnCommand	287
RenameEntry	288

Appendix D: Superagent Framework APIs **289**

DEadd	289
DEdelete	290
DEinit	291
DEinitNewObject	292
DEmodify	293
DEmodrtn	293
DEsearch	294

Appendix E: ODBC Wrapper Classes and Methods **301**

AttributeMap Class	302
Encoding Virtual Attribute Information	303
Encoding Table Information	304
AttributeMapIF Class	304
getSqlRowColumnCount	305
getSqlTableName	305
getSqlKeyColumnIndex	305
getSqlKeyColumnName	306
getLdapAttrIndex	306
getSqlColumnIndex	307
getLdapAttrName	308
getSqlColumnName	308
getSqlType	309
isMappedToSql	309
isMappedToLdap	310

isPrimaryKey	310
getLdapAttrFromSqlColumn	311
getSqlTypeBySqlColumn	311
isSqlColumnMappedToLdap	312
getSqlColumnFromLdapAttr	312
getSqlTypeByLdapAttr	313
isLdapAttrMappedFromSql	313
isLdapAttrMappedToSql	314
CTokenEx Class	314
Split	315
Join	316
CTableObject Class	316
UTFPack/MBCSPack	317
UTFUnpack/MBCSUnpack	317
Constructor Methods	318
OdbcWrapper	318
SQL Operations Methods	318
SqlInsertRow	319
SqlDeleteRow	320
SqlUpdateRow	320
executeSqlUpdate	322
Utility Methods	322
getLdapValueFromSqlColumn	323
getSqlValueForSqlColumn	324
escapeQuotes	325
getErrorMessage	325
getTableRowCount	326
getPackedRows	327

Appendix F: The Java IAM SDK 329

Installing the JIAM SDK	329
Setting Up the JIAM SDK	330
Sample Programs	330
JIAM SDK Reference	330

Appendix G: Custom Connectors 331

The JIAM Custom Connector	331
How to Define JIAM Custom Connector Extensions	332
How to Deploy JIAM Custom Connector Jar Files	333
Deployment on WebLogic or WebSphere	333
Deployment on JBoss	333

How to Validate Custom Connectors	334

Appendix H: Universal Provisioning Connector Reference	335

UPC Account Objects	335
Account Properties	335
Operational Properties	336
Virtual Properties	338
Deprecated Properties	339
Program Exit Types	340
Common Exit Types for Managed and Non-Managed Modes	340
Non-Managed Mode Exit Types	345
Service Level Agreement Monitor Exits (Non-Managed Mode)	350
Managed-Mode Exits	354
Non-Managed Mode Operations	357
Marking a Pending Operation As Completed	357
Remove an Account from the Directory	359

Index	361

Chapter 1: Provisioning SDK Overview

This guide provides detailed information about the CA Identity Manager Provisioning Server Software Development Kit (SDK). It is intended for developers who use one or more of the application programming interfaces (APIs) in the SDK to develop applications for managing connectors. It also discusses the construction of GUI plug-ins, parser tables, and connectors by CA development teams and by third parties. Developers must have C/C++ knowledge of building modules on standard MFC or Windows Intel platforms.

This section contains the following topics:

[How This Guide Is Organized](#) (see page 15)

[Development System Prerequisites](#) (see page 16)

[Provisioning SDK Installation](#) (see page 17)

[Provisioning Server Terminology](#) (see page 18)

[Provisioning Server Architecture](#) (see page 19)

[Provisioning SDK Components](#) (see page 28)

How This Guide Is Organized

The guide contains reference and instructional information covering the following Provisioning Server development topics and tasks:

Architectural Overview

"[Provisioning SDK Overview](#) (see page 15)" and "[Provisioning SDK Fundamentals](#) (see page 31)" contain background information to help you understand the Identity Manager Provisioning Server in a development environment.

Implementing a Custom Connector

A *connector* is the software that enables communication between a Provisioning Server and an endpoint system. This guide contains information about how to build and configure a custom connector.

See "[Implementing a Connector](#) (see page 45)" for an overview of the procedure.

Note: To establish a standard database connection, you can use the Connector Xpress, which lets you create and manage standard database connectors without the programming and system expertise required to create a custom connector. See the Connector Xpress online help for instructions.

Implementing an ODBC Connector

The SDK provides a sample ODBC-SDK connector, built on top of the CORE-SDK, that you can use as a template to develop new connectors for ODBC-accessible data.

The ODBC-SDK is built on top of the CORE-SDK. The ODBC-SDK is a pre-developed version of the CORE-SDK suitable for managing simple ODBC-accessible data.

See "[The Sample ODBC-SDK Connector](#) (see page 175)" and "[ODBC Wrapper Classes and Methods](#) (see page 301)" for information about developing an ODBC connection.

Note: ODBC connectors provide compatibility with previous versions of Identity Manager. Where possible, we recommend using JDBC connectors instead. You can manage users in database tables using a JDBC connector. See the ConnectorXpress documentation for details.

Developing Program Exits

The Provisioning SDK supports three different types of program exits: common program exits, Universal Provisioning Connector program exits, and native program exits in individual connectors.

See "[Common Program Exits](#) (see page 211)," "[Universal Provisioning Program Exits](#) (see page 235)," and "[Supporting Program Exits in Endpoints](#) (see page 255)" for information about developing program exits.

Development System Prerequisites

The Provisioning Server development environment should conform to the following requirements:

- Provisioning Server is installed. By default, the Provisioning Server is installed in the following directory:
 \Program Files\CA\Identity Manager\Provisioning Server
 - The Provisioning SDK is installed. By default, the SDK is installed in the following directory:
 \Program Files\CA\Identity Manager\Provisioning SDK
- Note:** The Provisioning SDK can be installed on a machine other than the Provisioning Server itself.
- JAVA SDK version 1.4.2 is installed. You must set the JAVASDK environment variable to point to the JAVA SDK location. This is required to build the JIAM Jar files for connectors.

- MS Visual C++ 2003, Version .NET, with MFC Shared Libraries for Unicode.
- Unicode libraries must be installed (MFC71u.dll)

You should have a working knowledge of the following:

- Developing and building C++ applications
- Developing and building Java applications
- Familiarity with the Identity Manager Provisioning Server

System Recommendations

A typical Provisioning Server development system on Windows, as recommended by CA Technical Services, has the following on three separate disk drives:

- The Microsoft Windows operating system (on drive C:)
- The Identity Manager Provisioning Server (on drive D:)
- Microsoft Visual Studio .NET
- The Provisioning SDK and custom connectors (on drive E:)

This is to optimize the combination of running a relational database (an Open Ingres database), an X.500 server eTrust Directory), and Provisioning Server on the same host. For RDBMS performance, the logging drive needs to be different than the data storage.

Provisioning SDK Installation

If you have not already done so, install the Provisioning SDK from your CA Identity Manager installation media.

To install the Provisioning SDK

1. Locate the CA Identity Manager Provisioning Components download or other media.
2. Start the Product Explorer.
3. Select Install Products, Developer Resources.
4. Select Provisioning SDK.
5. Follow the onscreen instructions to complete the installation.

Installation Directories

The following environment variable notation is used throughout this guide to refer to the Provisioning Server installation directories:

Variable Name	Default Installation Directory
<i>PSHOME</i>	\Program Files\CA\Identity Manager\Provisioning Server
<i>PSDK</i>	\Program Files\CA\Identity Manager\Provisioning SDK
<i>PSDKHOME</i>	\Program Files\CA\Identity Manager\Provisioning SDK\admin
<i>%DXHOME%</i>	\Program Files\CA\eTrust Directory\DXserver

Notes:

- *%DXHOME%* is a system environment variable defined when eTrust Directory is installed.
- *PSHOME*, *PSDK*, and *PSDKHOME* are notations of convenience only, which are used throughout this guide.

Provisioning Server Terminology

The *CA Identity Manager Programming Guide for Provisioning* uses the following important terms. These terms may have had different definitions in previous versions of CA products.

Here are some key terms used in this guide, and common equivalent terms:

connector

A *connector* is the software that enables communication between a Provisioning Server and an endpoint system. The code that makes up a connector can include a GUI plug-in, server plug-in, and agent plug-in.

A *dynamic connector* can be generated by Connector Xpress, and a custom *static connector* can be developed in Java or C++.

A connector is sometimes called a *namespace* or *option*.

connector server

A *connector server* is a Provisioning Server component that manages connectors. It can be installed on the Provisioning Server system or on a remote system.

There are two types of connector servers:

- The Java Connector Server (Java CS) manages connectors written in Java
- The C++ Connector Server (CCS) manages connectors written in C++

Note: In this guide, the C++ Connector Server is sometimes called the *SuperAgent*, a term used in previous versions of the Provisioning Server.

directory information tree

A *directory information tree* (DIT) is a tree of objects presented to clients by an LDAP server in which the clients can directly name and manipulate those objects by issuing LDAP commands.

endpoint

An *endpoint* is a specific target system or application, such as Active Directory or Microsoft Exchange, that is managed by the Java Connector Server or C++ Connector Server using a specific connector.

An endpoint is sometimes called a *directory*.

endpoint type

An *endpoint type* is a specific type of target system.

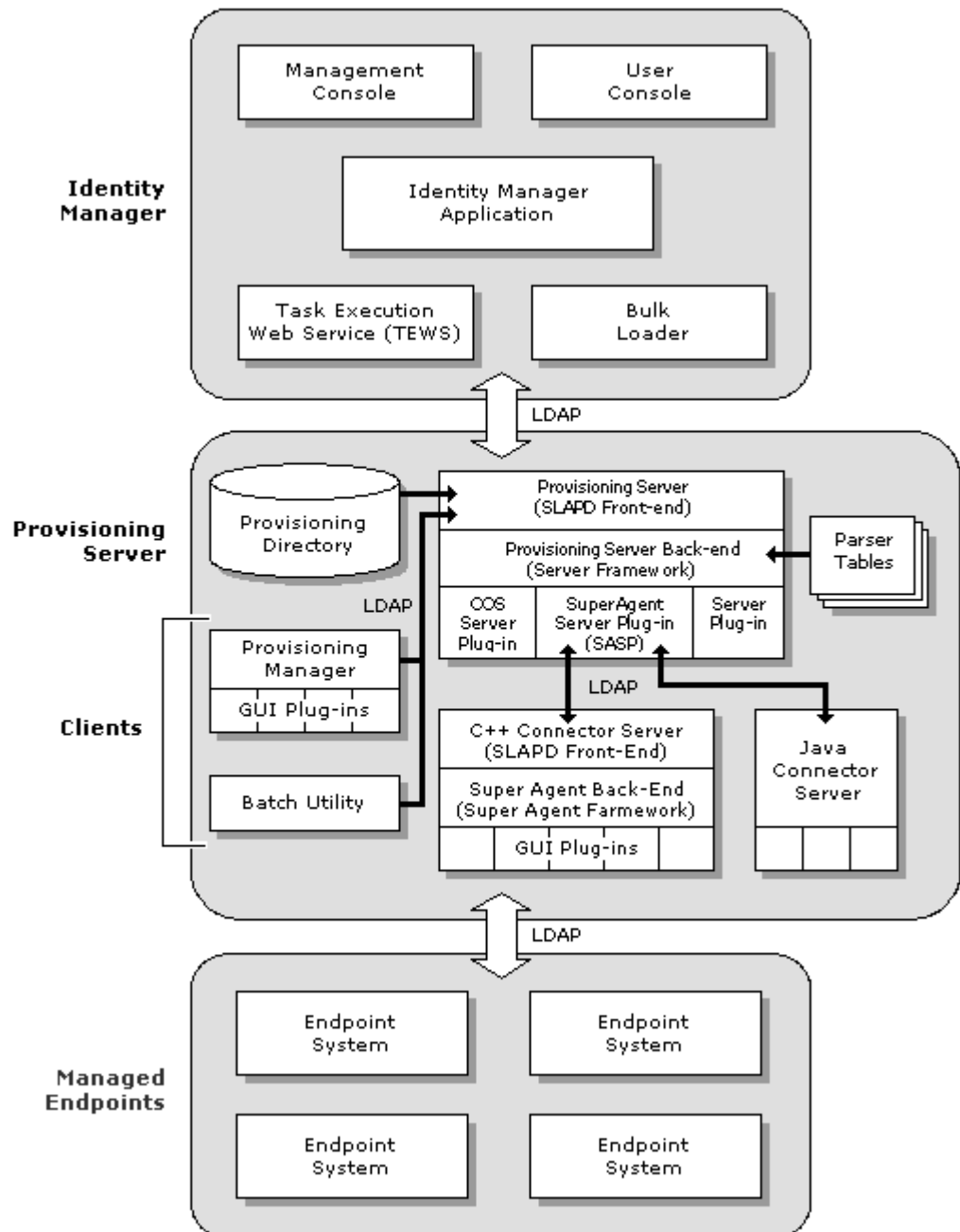
An endpoint type is sometimes called a *namespace*.

Provisioning Server Architecture

The Identity Manager Provisioning Server architecture consists of three main components:

- Identity Manager
- Provisioning Server
- Provisioning Server Clients
- Managed Endpoints

Depending on how it is configured, your Provisioning Server installation may have one or more of these modules. When adding new connectors to the Provisioning Server, you can develop each of the components in the appropriate module using the frameworks in the SDK.



Identity Manager

To address the escalating demands on IT, Identity Manager provides an integrated method of managing users and their access to applications, including:

- Assignment of privileges through provisioning roles. Specifically:
 - Roles that enable administrators to create and maintain user accounts
 - Roles that provide application rights to users
 - Roles that provision additional accounts to existing users (requires integration with the Provisioning Server)
- Delegation of the management of users and application access
- Self service options so users can manage their own accounts
- Integration of business applications with Identity Manager
- Options to customize and extend Identity Manager

Note: For more information, see the Identity Manager documentation.

Provisioning Server

The Provisioning Server consists of the following components:

- OpenLDAP (SLAPD) server configuration
- eTrust Directory configuration
- Provisioning Server back end to SLAPD
- Superagent back end to SLAPD
- Superagent Server Plug-in (SASP)
- Common Objects (COS) Server Plug-in
- C++ SDK Sample Connector
- Program Exit samples

The Provisioning Server acts as the manager and command center for all communication. The Provisioning Server is based on the OpenLDAP stand-alone directory server called SLAPD.

SLAPD servers have two components: a *front end* and one or more *back ends*.

- The front end processes LDAP requests from the client interfaces and passes those requests on to back ends.
- The back ends implement separate directory servers and manage a separate portion of the SLAPD server's Directory Information Tree (DIT).

The Provisioning Server includes the following SLAPD back ends:

- Provisioning Server back end
- SuperAgent back end

Server plug-ins are runtime libraries that do the following:

- Accept and validate requests from client interfaces
- Search, read, and store information in the Provisioning Directory
- Issue requests to connectors which then communicate with endpoint systems. Connectors run within the C++ Connector Server or Java Connector Server.

OpenLDAP (SLAPD) Server Configuration

The OpenLDAP (SLAPD) product allows easy implementation of LDAP servers. SLAPD is an implementation of a generic LDAP Server. By configuring it, you can instruct it to load plug-in back-end libraries that implement different LDAP servers, each of which is shielded from protocol specific knowledge about how data is transferred over the LDAP protocol.

Note: OpenLDAP is not a component of the Provisioning Server, but is included in the Provisioning Server installation with configuration files that load the eTrustadmin and superagent back ends. (As used in this case, eTrustadmin and superagent are literal backend names.)

eTrust Directory Configuration

The eTrust Directory provides both the Provisioning Directory used by the Provisioning Server back end for its persistent data storage needs, and provides the directory routers necessary to support various configurations. By using routers wherever LDAP connections are used to connect components, one can take advantage of the replication, load-balancing, failover, and firewall tunneling features of eTrust Directory.

The Provisioning Directory is an LDAP directory in which the Provisioning Server stores all its global security data. This data includes:

- Provisioning data (for example, global users, provisioning roles, account templates, configuration)
- Endpoint data (objects representing endpoints, accounts and other objects) that is populated in the Provisioning Directory via a process called acquiring endpoints.
- Operation data used for reporting operation status details to requesting clients.
- Notification data (queued notifications which are fed to the Identity Manager via the inbound notification feature)

The Provisioning Server works with many different LDAP-enabled directories. By default, the Provisioning Server installs eTrust Directory as its default Provisioning Directory. Although Provisioning Directory is not a component of the Provisioning Server, it is included in the Provisioning Server installation with configuration files that control how the Provisioning Server accesses its Provisioning Directory and how LDAP communications between components.

Note: For more information, see the *High-Availability Configuration Guide*.

Provisioning Server Back End to SLAPD

The main back end is known as the Server Framework, or Server Plug-in Framework. The Server Framework implements server behavior that is common across all connectors using run-time DLLs called server plug-ins.

This component is contained in the following DLLs:

- eTrustAdminServer.dll—Registered back-end DLL.
- EtaServerMsg.dll—English-language customer messages, a DLL that can be replaced by other language message DLLs during a localization process using message translators.

Each installed connector includes a binary parser table, which defines the name of that connector's server plug-in DLL. Two of those DLLs that are installed with the core server are the COS and SASP server plug-ins.

More information:

[Superagent Server Plug-in \(SASP\)](#) (see page 24)

[Common Objects Server \(COS\) Plug-in](#) (see page 27)

Superagent Back End to SLAPD

The Provisioning Server has a superagent backend, which is loaded into the C++ Connector Server (CCS), a separate SLAPD process from the Provisioning Server. The CCS is used by all new C++ connectors and provides an object-oriented application framework.

The CCS simplifies C++ connector development by doing the following:

- Shields you from LDAP protocol-specific details. The CCS keeps the necessary amount of LDAP knowledge to a minimum.
- Minimizes LDAP deployment overhead.
- Standardizes the set of functions that a connector needs to implement, which coincides with the simple LDAP functional model.
- Loads connector DLLs dynamically through the LDAP ADD operation. This feature provides the flexibility to add new connectors without having to recompile or restart the superagent.

- Provides dynamic DIT support, allowing flexible container structures within endpoints.
- Provides a synchronization mechanism for connectors that must run in a single-threaded environment.
- Provides logging support.

By writing connectors using the CCS, you write only the code that is specific to managing an endpoint type. The CCS provides utilities for debugging, reporting, and fast prototyping.

You can configure the Provisioning Server to have additional C++ Connector Servers and Java Connector Servers. This can increase parallelism by dedicating connector servers to particular endpoints or endpoint types.

Superagent Server Plug-in (SASP)

Several server plug-ins exist in the Server framework. The most common server plug-in is called the Superagent Server Plug-in (SASP). SASP knows how to send LDAP requests to the Provisioning Directory as well as to the C++ and Java Connector Servers. SASP uses parser tables to indicate how to process requests from the client interfaces.

Every connector supplies a server plug-in DLL that provides C++ objects to manage the object classes of the endpoint type. Most connectors use the pre-defined SASP server plug-in DLL (SASPRegister.DLL) as their server plug-in.

The SASP server plug-in, like all server plug-in DLLs, registers two C++ objects for each class of object that it manages:

C++ Manager Object

This module processes incoming LDAP requests (Add, Modify, Search, and so on) against objects of this object class.

When LDAP requests are received by SLAPD, they are sent to the Provisioning Server back end, which determines what kind of object is being operated upon (what object class), locates the manager object for that object class and sends the request to the appropriate method (Add, Modify, and so on) of that manager object for processing.

For SASP, all of the implementation for manager objects is in the core server (eTrustAdminBackend.DLL), in the C++ Manager base class (EtaMgr) or one of its derived classes (EtaDirectoryMgr, EtaPolicyMgr, EtaAccountMgr, EtaContainerMgr). This base implementation uses parser table data to automatically handle the differences between different object classes.

Server plug-ins override various behaviors where the parser table language is insufficient to describe all of their required behaviors. Thus, some connectors provide custom server plug-ins that clone `saspRegister.DLL`, replacing one or more of the C++ objects it creates with connector-specific objects from derived C++ classes. The derived classes override only those virtual functions that are necessary to redefine the standard behavior to meet their objects requirements.

C++ Agent Dispatcher Object

This module sends LDAP requests (such as Add, Modify, and Search) to a managed endpoint to carry out requests to manipulate the real object being managed by the Provisioning Server.

In general, the agent dispatcher is C++ code running in the Provisioning Server to communicate with the endpoint. However, the recommended way is to write a connector running in a connector server rather than an Agent dispatcher running in the Provisioning Server.

For most connectors, the agent dispatcher object is simply one of the objects created by the default SASP implementation. For these classes, communication to the managed endpoint is done by sending LDAP requests to the connector servers.

SDK Sample C++ Connector

While technically not part of the Provisioning Server, the SDK sample C++ connector shows how to add custom connectors to the Provisioning Server. In particular, the SDK sample C++ connector can be used to manage accounts and groups, and exercise basic functionality like user provisioning.

Program Exit Samples

The DLL and SOAP Program Exit samples, while also not technically part of the Admin Server, provide a means to exercise the common program exit mechanism that otherwise would not be testable without specifically writing those program exits.

Provisioning Server Clients

A *client* is any computer that requests work from the Provisioning Server. These requests can be sent in many different forms, depending on the Provisioning Server software installed.

Client interfaces are programs running on the client; they consist of the following:

- **Provisioning Manager**—Used by administrators, the Provisioning Manager is a graphical user interface (GUI) that organizes provisioning tasks into specific groups.
- **Batch Utility**—The command line interface that administrators use to process large amounts of information. It is capable of performing every task that the Manager GUI handles.

Client Interfaces

The client interfaces communicate with the Provisioning Server using an LDAP protocol. The Provisioning SDK provides API functions that let you extend the Provisioning Manager and Batch Utility without having LDAP-specific knowledge.

Administrators can use these interfaces to add, modify, or delete user objects and their properties from any Windows workstation or server in the enterprise.

Provisioning Manager

You can extend Provisioning Manager to administer additional endpoints by writing GUI plug-ins and parser tables.

A GUI plug-in is a runtime loadable DLL that contains icons, menus, property pages, and logic that manages global users, accounts, and other objects in the endpoint type.

A parser table is a textual definition of the objects in the endpoint type. The client interfaces and the Provisioning Server use the parser tables to understand how to process the objects in the endpoint.

Batch Utility

You can extend the Batch Utility by writing a parser table only. No GUI plug-in is required.

In addition to extending the Batch Utility, you can write your own LDAP client without using the APIs provided in the SDK.

Managed Endpoints

An endpoint type is a specific type of target system. There is typically one agent plug-in for each endpoint type, but there can be multiple endpoints for each endpoint type.

For each endpoint type that you want to manage, you must have an agent plug-in. Agent plug-ins are DLLs that are loaded into the object-oriented superagent framework. Agent plug-ins are responsible for representing each of the object classes in your endpoint type, and translating Add, Modify, Delete, and Search operations on those objects into corresponding actions against the Connector Server.

Common Objects Server (COS) Plug-in

Every object class managed by the Provisioning Server must belong to an endpoint type representing a set of related managed endpoints (Oracle for example). The managed object classes represent objects specific to that managed endpoint (Users, Roles, Profiles). However, all the pre-defined object classes like Provisioning Roles, Global Users, and Admin Profiles that exist in the Provisioning Directory in all Provisioning Server installations before any connectors are installed must also belong to an endpoint type.

We call that endpoint type a Common Objects Server (COS). COSRegister.DLL contains the server plug-in for the COS endpoint type. As with all server plug-in DLLs, this DLL registers two C++ objects for each class of object that it manages:

C++ Manager Object

This module implements special behaviors associated with the respective common objects.

The user provisioning functionality associated with global users and provisioning roles, for instance is largely implemented in the manager objects for these two object classes (implemented by C++ classes CosGlobalUserMgr and CosRoleMgr).

C++ Dispatcher Object

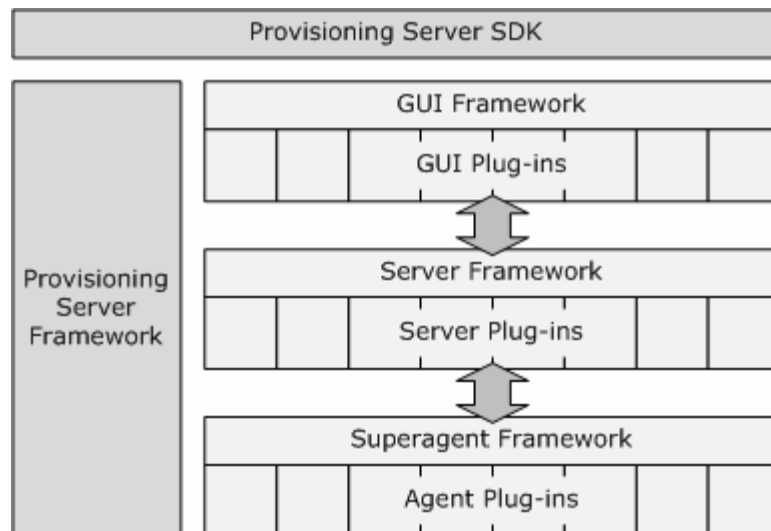
This object is never called because common objects do not have a corresponding underlying agent object to manage.

Provisioning SDK Components

The Provisioning SDK lets you customize Provisioning Server architecture. You can use the SDK to create and manage additional connectors. The SDK has been structured into the following main components:

- GUI Framework
- Server Framework
- Superagent Framework

When you add new connectors to the Provisioning Server, you may create a GUI plug-in to extend the GUI Framework and create an agent plug-in to extend Superagent framework. You do not need to create a new server plug-in. You can use the SASP as the server plug-in for the new connector.



GUI Framework

The GUI Framework component helps you build GUI plug-ins. GUI plug-ins represent the Provisioning Manager, and are responsible for displaying new connectors and their objects.

Server Framework

The Server Framework is the component that enables GUI plug-ins and connectors to communicate. The Server Framework is composed of several server plug-ins. All new connectors use the Superagent Server Plug-in (SASP).

This plug-in has the following functionality:

- Accepts and validates requests from the GUI plug-ins.
- Searches, reads, and stores information in the Provisioning Directory
- Issues requests to the appropriate connector.

Superagent Framework

The Superagent Framework helps you build connectors using the superagent. Connectors are responsible for communicating with the superagent and the connector server.

The Superagent Framework functionality is as follows:

- Maintains a combined DIT of all of the connectors as nodes in an in-memory DIT data structure
- Processes incoming LDAP requests, determines which object in the DIT is being manipulated by the LDAP request, and calls the appropriate connector object's methods to perform the desired function
- Supports dynamic DITs that let you manage connectors with a container hierarchy that can change over time as containers are added or deleted

By using the Superagent Framework, development effort is simplified because you write only the code that is specific to managing the connector. In particular, you do not need to know the LDAP-specific details, such as parsing the DN that is associated with an incoming LDAP request and determining what operation should be performed on the endpoint object.

Chapter 2: Provisioning SDK Fundamentals

This section contains the following topics:

[The LDAP Protocol](#) (see page 31)
[Objects in an LDAP Directory](#) (see page 32)
[Parser Tables](#) (see page 35)
[Directory Information Trees \(DITs\)](#) (see page 38)
[Processing LDAP Requests](#) (see page 40)
[Code Portability](#) (see page 41)
[Converting Program Exit Code](#) (see page 43)

The LDAP Protocol

The Provisioning Server uses LDAP (Light Directory Access Protocol) to store persistent information and to communicate between the different components. The LDAP protocol is designed to manipulate hierarchical, object-oriented data.

The primary unit of storage for an LDAP protocol is an *object*. An object represents items such as an account, a directory, or an intermediate container. All these objects are organized in a Directory Information Tree (DIT). A DIT defines the structure or schema of your endpoint to the Provisioning Server.

LDAP Operations

The LDAP protocol provides a simple set of operations that LDAP clients can perform on objects. These operations are the following:

- **ADD**—Adds a new object to the DIT.
- **MODIFY**—Modifies an object in the DIT.
- **SEARCH**—Locates or enumerates one or more objects in the DIT.
- **COMPARE**—Compares an object in the DIT to a certain criteria.
- **MODRDN**—Renames an object by modifying its Relative Distinguished Name (RDN).
- **DELETE**—Deletes an object from the DIT.

LDAP Directories

When an endpoint presents information to LDAP Clients through a DIT, it is called an LDAP directory. The Provisioning Server has three LDAP directories:

Provisioning Server

Presents a DIT to the client interfaces containing objects for each item in a directory and for all common object items. They can read or write objects to the DIT using the LDAP protocol.

Provisioning Directory

Can be an external directory, such as eTrust Directory. The Provisioning Directory permits objects in its DIT to be read or written by LDAP clients or the Provisioning Server.

Connector Server

Presents a DIT to its client that contains all the objects presented by the connectors when they are loaded. Typically, the client is the SASP running in the Provisioning Server. SASP uses LDAP operations to read or write objects to the Connector Server DIT. These operations can modify the endpoint managed by the connectors.

Objects in an LDAP Directory

An LDAP directory can have several types of objects. To organize the different types of objects in a DIT, you must verify that each object belongs to a class. An object class is a generic specification for a group of objects sharing certain characteristics.

When you create an object in a directory, you specify an object class for the object. The object is referred to as an instance of the object class. The object class that you select determines how the object is classified in the directory, specifically:

- What kind of information do you store about the object
- Where the object is located
- How the object is named
- How the object relates to other objects

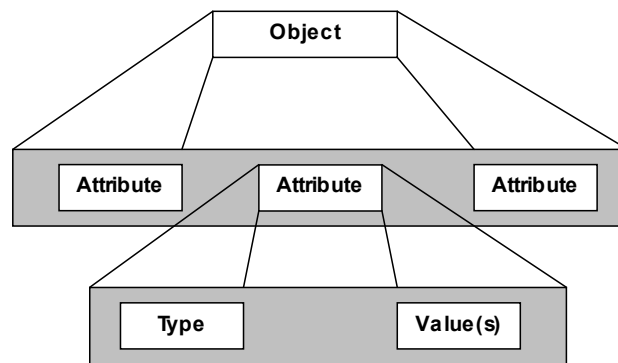
For example, the SDK Accounts object is a container that groups all explored accounts. The SDK Accounts object belongs to the eTSDKAccountContainer object class.

Object Attributes

The items of information stored in an object are known as *attributes*. Attributes identify objects and you use them when searching for an object. An attribute is defined by the following:

- One or more values
- A data type

The following diagram illustrates how attributes define an object:



For example, the Account Properties tab in the SDK Account property sheet displays the following attributes for an account object:

- Login Name
- City
- Employee Id
- Password

The object class to which the object belongs determines the object's attributes. The object class provides a list of possible attributes. Some attributes are mandatory, that is, they must appear in an object of a particular object class. Other attributes are optional. Mark mandatory attributes with an asterisk (*) in the property page for the object.

Most attributes are *single-valued*. Single-valued means that each object in the class has one value for the attribute. An example of a single valued attribute is the naming attribute, which uniquely identifies the object.

In some cases, the attribute can be *multi-valued*. Multi-valued means that the object can store an unordered list of values for the attribute. An example of a multi-valued attribute is a group membership attribute in an account object. The attribute has one value, the group name, for each group to which the account belongs.

The object's attribute type refers to the name of the attribute and the data type. If two objects have attributes with the same name, the objects must have the same data type. For example, the SDK Directory uses a string attribute named `eTSDKHost`. When another object class in the DIT wants to define an `eTSDKHost` attribute, the object class must define it as a string attribute. If you define conflicting data types with an attribute, an error message is issued.

Note: String attributes are encoded using the character set defined in UTF-8.

Distinguished Names

You identify each object in a connector by its *distinguished name* (DN), which has the following format:

`attribute3=value3,attribute2=value2,attribute1=value1`

attribute1

Represents an object at the top of the tree

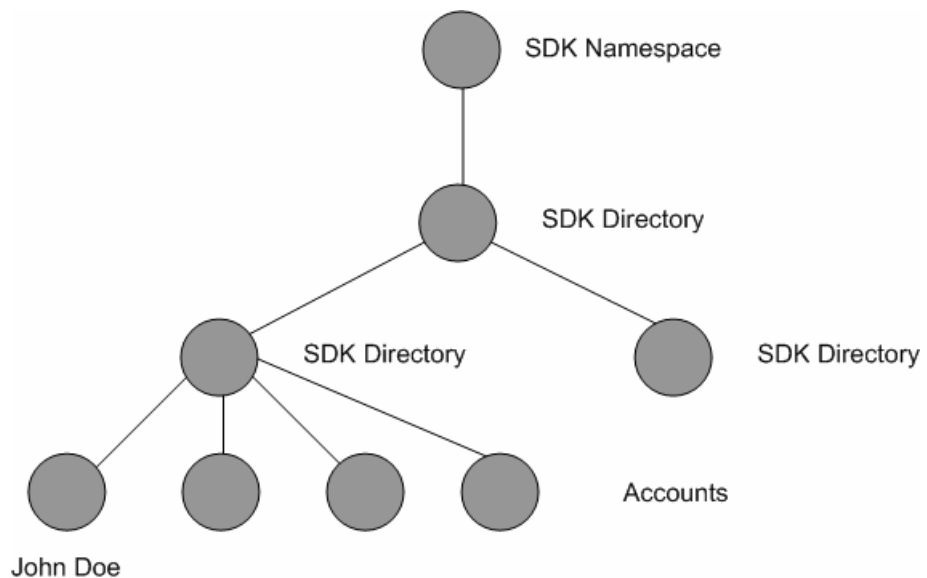
attribute2

Represents a child for that object

attribute3

Represents the child of that object

The DN of an object is the concatenation of the *relative distinguished name* (RDN) of the object and all its ancestors. For example, consider the following SDK Namespace DIT:



The RDN of one account object would be:

eTSDKAccountName=John Doe

The DN of this account would be:

eTSDKAccountName=John Doe,eTSDKAccountContainerName=SDK
Accounts,eTSDKDirectoryName=SDK Directory, eTNamespaceName=SDK
Namespace,dc=domain_name,dc=eta

Note: dc=eta is the root DN or DN suffix for the Provisioning Server DIT. The *domain_name* represents your domain name.

Parser Tables

A *parser table* is a textual definition of the objects in a endpoint type.

Each endpoint type has one parser table associated to it. The Provisioning Manager uses parser tables to manage objects.

Creating a parser table involves the following basic tasks:

- Defining the schema
- Defining objects in a parser table
- Generating schema files

Note: The parser table is accessible to the GUI plug-in and the Provisioning Server only. It is not accessible to the Connector Servers and connectors. When building a new connector, the attributes, objects, and organization must be implemented consistently in the connector as it is specified in the parser table. Failure to do so results in communication problems between the Provisioning Server and the connector.

Schema Definition

A schema is the set of rules that define the objects, attributes, and object relationships in a directory information tree (DIT).

You define a schema using a parser table. By defining a schema, you ensure that the client interfaces and Provisioning Server process the objects in your endpoint correctly. OpenLDAP and eTrust Directory require schemas to be defined in alternate file formats.

The Provisioning Server includes a SCHEMAGEN utility to create these alternate schema representations from the parser tables.

Creating a Parser Table

You define a schema by creating a parser table. Parser tables contain more information than would be contained in a schema file. This information includes the following:

- Descriptions of the object classes, attributes, and relationships
- DLL names for your GUI plug-in, server plug-in, and agent plug-in
- File names for icons
- Organization of your objects
- Information about the behaviors of the objects in the endpoint type
- Constraints for dynamic endpoint organization

The source for parser tables consists of keywords, parser table values, comments, and `#include` statements.

To create a parser table:

1. Define the endpoint type.
2. Define the classes in the endpoint type.
3. Define the attributes that make up the objects of each class.

More information:

[Building Parser Tables](#) (see page 105)

The SCHEMAGEN Utility

The Provisioning SDK contains a SCHEMAGEN utility program, which reads all parser tables and generates schema files for each endpoint in an endpoint type. Since the Provisioning Server automatically writes the schema files for each supported endpoint, you only need to know how to write parser tables.

The SCHEMAGEN utility program is located in the *PSHOME/bin* directory, if the SDK is installed with the Provisioning Server. The output directory is *PSHOME/data*.

If the SDK is installed stand-alone, the location is the *PSDKHOME/bin* directory, and the output directory is *PSDKHOME/data*.

You must run SCHEMAGEN for each newly created endpoint type.

To generate schema files with SCHEMAGEN

1. Copy the XXXParse.ptt file to *PSHOME/data*, where XXX is a three-letter abbreviation for your endpoint type.
2. Open a DOS screen and change the directory to *PSHOME/bin*.
3. If the SDK is not installed on the same machine as the Provisioning Server, copy the eTrust_XXX.schema file from *PSDKHOME/data* to *PSHOME/data*.
4. Run the following command:

```
SCHEMAGEN /GENOID -n XXX:oid_prefix_number
```

XXX is a three-letter abbreviation for your endpoint type.

oid_prefix_number is the OID number that you select for your connector and should be greater than 10,000.
5. Move the etrust_XXX.schema file from *PSDKHOME/data* or *PSHOME/data* depending on whether SDK is stand-alone or on the same machine as the server to *PSHOME/data*.
6. Edit *PSHOME/data/etrust_admin.conf* so it includes the following line:

```
include "C:\\Program Files\\CA\\Provisioning  
Server\\Data\\etrust_XXX.schema"
```
7. Stop and then restart the Provisioning service.
8. Move the etrust_XXX.dxc schema file from *PSDKHOME/data* or *PSHOME/data* depending on whether SDK is stand-alone or on the same machine as the server to *%DXHOME%/config/schema*.

9. Edit `%DXHOME%/config/schema/etrustadmin.dxc` so it includes the following line:

```
source "etrust_XXX.dxc";
```

This line should be similar to the existing lines.

10. Reinitialize the eTrust Directory etrustadmin Service by entering the following command:

```
dxserver init etrustadmin
```

Note: You do not need to perform these steps if you are configuring the SDK Sample Connector. Instead, copy `SDKParse.ptt` to `PSHOME/data` and restart the Provisioning service. The Provisioning Server installation already provides the schema files for the SDK Sample Connector.

Directory Information Trees (DITs)

Typically, multiple LDAP servers are bundled together by linking their back ends into an OpenLDAP SLAPD server. Even though these servers are linked, they are independent of each other. Because of this, SLAPD must have a way to distinguish between them. In the Provisioning Server, the SLAPD server distinguishes between back ends by matching the DN in an LDAP request to the DN suffix of the back end.

In the Provisioning Server, a separate LDAP server manages each DIT.

Each back end in the Provisioning Server has one of the following DITs:

- Provisioning Server DIT
- Provisioning Directory DIT
- Connector Server DIT

Provisioning Server DIT

The Provisioning Server DIT contains the common objects and endpoint that are provided with the Provisioning Server. You can view all the objects and attributes in the Provisioning Server DIT through the Provisioning Manager or Batch Utility.

Provisioning Directory DIT

The Provisioning Directory DIT provides a subset of the objects and attributes in the Provisioning Server DIT. The Provisioning Directory DIT keeps all the common objects, as well as simplified copies of the connector-specific objects. The simplified copies contain the object name and any extended attributes, such as statistic attributes. The Provisioning Server DIT refers to the Provisioning Directory DIT for the objects and attributes that it stores.

Connector Server DIT

The Connector Server DIT provides another subset of objects and attributes in the Provisioning Server DIT. The Connector Server DIT contains all connector DITs. Each connector DIT must define its objects. This includes any necessary objects that provide the desired organizational structure.

Identifying DITs

Each managed endpoint sends information to each DIT. These DITs are very similar, and differ only in the subset in which the object classes and attributes in the endpoint type are defined. You can send information to a DIT by loading the associated back end.

To load the appropriate back end for the DIT, build an LDAP request using the DN suffix appropriate for the DIT.

Back End	DN Suffix	Usage
Provisioning Server	dc= <i>domain_name</i> , dc=eta	Use if you are accepting or validating requests from the client interfaces
Provisioning Directory	dc= <i>domain_name</i> , dc=etadb	Use if you are searching, reading, or storing information in the Provisioning Directory
Connector Server	dc= <i>domain_name</i> , dc=etasa	Use if you are issuing requests to endpoints through the Connector Server

By connecting to each of these back ends using an LDAP-enabled browser, you can view each DIT. The hierarchy of the objects in each DIT is consistent. The Connector Server DIT, however, does not include the Common Objects connector or any other managed endpoint that does not use the Connector Server.

Defining DITs

You define the DITs in your parser table. The parser table defines the types of information that the managed endpoint contributes to each DIT. The `DATALOCATION` property indicates the DIT where the information is stored.

More information:

[Building Parser Tables](#) (see page 105)

Processing LDAP Requests

When a client interface sends a request to the SLAPD server, the request always includes two items of information:

- The type of the request
- The DN of an object being accessed by the request

In this example, a modify request is sent from the client interface for an object with the following DN:

```
eTSDKAccountName=John Doe,eTSDKAccountContainerName=SDK  
Accounts,eTSDKDirectoryName=SDK Directory, eTNamespaceName=SDK  
Namespace,dc=domain_name,dc=eta
```

This is a high-level view of the request handling process:

1. The SLAPD server receives this request and reads the DN. The DN ends with the suffix of the Provisioning Server back end, `dc=eta`. This indicates to the SLAPD server to send the request to the Provisioning Server back end. The domain component, `dc=domain_name`, identifies to which Provisioning Server this request should be routed.
2. When the Provisioning Server back end receives this request, it reads the RDN of the object and uses the RDN to identify the object class in the parser table. In the example above, the RDN is `eTSDKAccountName=John Doe`. The only object class in the parser table whose naming attribute is `eTSDKAccountName` is the `eTSDKAccount` object.
3. Once the object class is identified, the Provisioning Server back end reads the parser table to identify important information, performs validations, and performs data conversions. The Provisioning Server back end is aware that the server plug-in defined for the SDK Connector is `saspRegister.dll` by reading the parser table.

4. When the Provisioning Server back end finishes reading the parser table, performing validation, and converting data according to the parser table, it passes the request to SASP. SASP, used by all new connectors, sends LDAP requests to the Provisioning Directory and to the Connector Server to complete each request.
5. SASP uses the parser table to determine how to process the request and determines which attributes and objects are maintained in the Provisioning Directory and which are maintained on the endpoint system. While processing the request, SASP may need to send LDAP requests to the endpoint, to the Provisioning Directory, or to both, depending on the nature of the request.
6. The LDAP request that SASP needs to send depends upon which attributes are being modified. If the parser table identifies some or all of the attributes being modified as being stored on the endpoint system, an LDAP modify request is sent to the endpoint through the Connector Server and connector to change those attributes. In any case, an LDAP Modify request is sent to the Provisioning Directory to update the statistics attributes and any other attribute that is stored in the Provisioning Directory, as indicated by the parser table.
7. To complete the request, the success status is reported back, through the Provisioning Server back end, to the SLAPD server, and finally to the client interface that initiated the request.

Code Portability

Since the Provisioning Server supports UNIX (Solaris) as well as Windows, CA recommends that you write portable code, thus allowing your connector to run on both platforms.

The components that run in the Provisioning Server or connector servers are the only components that may need portable code. The main modules involved are part of the superagent plug-in.

The GUI plug-in code does not need to be portable, as it can only be used by the Provisioning Manager, which runs exclusively on Microsoft Windows platforms.

Factors Affecting Portability

The major factors that affect portable code are the following:

- Target operating system
- C++ compiler
- Available supporting C++ libraries

The set of C++ compilers that are currently supported is the following:

- Microsoft Visual Studio.NET 2003
- Sun Studio 10 or later

Header Files

The Provisioning SDK provides a set of header files that can be used to hide the platform-specific details. The header files can be included in the `include/tools` directory, and will fit both platforms, as shown in the following examples:

- By including `#include tools/os.h` in your code, you can include an OS specific header file without having to add `#ifdef` macro into your code to include the Windows or Solaris specific header, since this is done for you by `tools/os.h`.
- You can write portable code without having to include platform conditional compilation.

Note: UNIX filenames are case-sensitive. For this reason, you must consider the case when supplying the filename in statements, such as the `include` statement.

Conditional Compilation

If you need to include conditional compilation in your code, use the following macros:

```
#ifndef _MSC_VER

    //Visual C++, for example, Windows

#else

    //UNIX

#endif
```

Portable `printf()` Processing

Windows and UNIX handle `printf` and `wprintf` differently.

Windows

- `printf("%s", mbcs_string);` prints a MCBS or ASCII string.
- `wprintf("%s", wide_string);` prints a `wchar_t` string

UNIX

`printf()` and `wprintf()` process `"%s"` in the same way, and in both cases, it expects an MBCS/ASCII string.

Portable Code

Wide strings should be explicitly identified with `"%ls"`, which works in the same way on both platforms. The following statement can handle both MBCS and wide strings:

```
printf("%s,%ls",char_string,wide_string)
```

Windows and UNIX handle NULL differently:

Windows

When NULL is passed to `printf`, Windows handles it, and avoids a crash.

UNIX

When NULL is passed to `printf` on Solaris, a crash is likely to occur.

Portable Code

Avoid passing NULL to any routine that expects a non-NULL `char*` value.

Converting Program Exit Code

New program exits should use the new portable, Xerces-based, XML library, `uetaxml`, instead of the old `EtaXml` library.

The new `uetaxml` library has the same interface as the old `EtaXml` library. You may want to convert your existing program exits to the new `uetaxml` library. Since the interface is the same, you do not need to modify the code. Simply, recompile your code, and link it to the new `uetaxml` library instead of the old `EtaXml` library.

Chapter 3: Implementing a Connector

This section contains the following topics:

[Connector Overview](#) (see page 45)

[Connector Design Specification](#) (see page 46)

[Implement a Custom Connector](#) (see page 48)

Connector Overview

An *endpoint type* is a specific type of target system that the Provisioning Server connects with. A *connector* is the software that enables communication between the Provisioning Server and an endpoint system.

You can use the Identity Manager Connector Xpress to create and manage connectors without the technical expertise required to create a custom connector. The Connector Xpress wizard lets you create, configure, map tables and groups, review metadata, and register the connector to be managed by the Provisioning Server.

Note: See the Connector XPress online help for more information.

Connector Xpress works with a special kind of endpoint type called a dynamic endpoint type, which can be serviced by a single generic connector for a family of related endpoint types (JNDI for directory-based connectors, JDBC for database-based connectors).

A dynamic connector has XML metadata that is generated and maintained by end-users using Connector Xpress, whereas a static connector has metadata that is generated and maintained by developers.

This guide contains information about how to develop and configure a static custom connector in C++.

Connector Design Specification

Before starting to develop your connector, CA recommends that you observe standard project procedure and write a Detailed Design Specification (DDS) that includes the following information.

Functional specifications:

- The type of endpoint system this connector will manage
- The types of endpoint schema objects this connector will manage
- The operations the custom connector performs on the schema objects (such as add, modify, delete, etc.)
- How to present object information to operators who use the Provisioning Server
- How to initiate the operation for each object and what are the expected results
- How the connector keeps the endpoint system and Provisioning Server synchronized

Schema-related specifications:

- The 3 letter acronym that uniquely identifies the endpoint in the parser table
- A definition for each type of managed object
- A definition for the endpoint policy object
- Definitions for account attributes that are synchronized with the policy

GUI plug-in specifications:

- Definitions of C++ classes and GUI dialogs for each managed object type
- Definitions of the GUI dialogs of custom connector policy object

Connector specifications:

- How to explore each object type
- How to implement the operations that act upon each object type

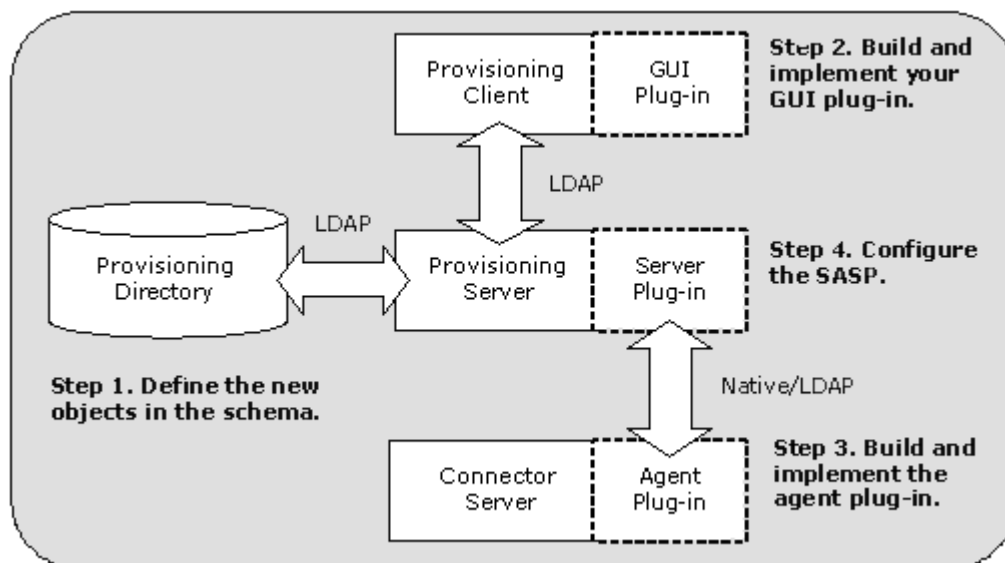
The DDS should also address the following issues:

- Remote agent implementation (if necessary)
- Unit test plans and expected results for each operation
- Packaging and deployment of connector executable code

Implement a Custom Connector

Implementing a connector entails the following steps:

1. Define new objects in the schema.
2. Build and implement the GUI plug-in.
3. Build and implement the agent plug-in.
4. Configure the server plug-in.



Note: The SDK sample connector is referred to throughout the *Programming Guide for Provisioning*. You should build and compile the SDK sample connector before you implement a custom connector.

More information:

[The Sample Connector](#) (see page 55)

Define Objects in the Schema

When you implement a connector, the first task is to define objects in the schema.

For example, if you scan the container tree in the Directory Content dialog, you can identify the schema of the SDK Namespace. You can also use a directory browser (such as JXPlorer) or the ShowPTTDIT utility to see this information.

Starting from the bottom, the following objects appear:

John Doe account object

Account objects are called leaf objects. A leaf object does not have any objects below it. All leaf objects belong to an object class.

SDK Accounts container object

Member of the eTSDKAccountContainer object class.

Account containers contain all the account objects that exist in the directory.

SDK Directory container object

Member of the eTSDKDirectory object class.

In the example above, the SDK Directory container object contains an SDK Accounts container object and an SDK Groups container object.

The directory container is the top-level node, and all directory objects appear under the top-level node.

SDK Namespace container object

Member of the eTNamespace object class.

Namespace containers are top-level containers. They contain all the objects in the endpoint type.

For new connectors, define similar object classes to those defined in the SDK Namespace. While the parser table syntax lets you define many possible schemas, you should follow some rules when creating the objects for your connector. These rules make it easy to integrate the connector into the Provisioning Server.

More information:

[Parser Tables](#) (see page 35)

[Schema Definition](#) (see page 36)

[Building Parser Tables](#) (see page 105)

Build and Implement the GUI Plug-in

To build and implement the GUI plug-in, the major tasks are:

1. Define your endpoint class, directory class, and class attributes for the parser table.
2. Create the dialogs and property sheets for your endpoint type.
3. Verify that the dialogs and property sheets work properly and check the controls on the fields.
4. Package your dialogs and property sheets into a DLL and identify the DLL to Provisioning Manager.

Notes:

- The SDK Connector GUI plug-in DLL is CASDKGUI.DLL.
- The PSDKHOME/Samples/SDK/GUI directory implements all dialogs, property sheets, and parser tables.
- You need the source of your SDK dialogs, the parser table, and the script files to instantiate the SDK Namespace.
- RB: The property pages developed for manipulating the account object class is often used for manipulating the policy object class, so input validation may be different between property sheets using the same pages.

More information:

[Building a GUI Plug-In](#) (see page 163)

Build and Implement the Connector

To build and implement the agent plug-in, the major tasks are:

1. Define your endpoint class, directory class, and class attributes to the parser table.
2. Define C++ objects representing each of the object classes of your endpoint type.
3. Define methods to implement LDAP operations, such as add, modify, search, or delete on the objects you defined in Step 1.
4. Package the objects into a DLL named XXXXNamespace.DLL, which the superagent framework loads.

Notes:

- The SDK Connector DLL is SDKNamespace.DLL.
- The PSDKHOME/Samples/SDK/Agent directory implements the SDK Connector.
- Verify that the schema and DIT for the connector is structured the same way as the GUI plug-in parser table. If these are not structured the same way, communication problems will occur between the Provisioning Server and the client interface.

More information:

[Building an Agent Plug-in](#) (see page 163)

Configure the Server Plug-in

Specify how to explore each object type.

Specify how to programmatically implement the operations that act upon each object type.

To configure the superagent server plug-in, you edit your parser table as described:

1. Define SASP as the server plug-in for your connector:

```
NAMESPACE your namespace
    smplugindll=SASPRegister
```

When you want to add a new connector, you must set the smplugindll property. If you use the SDK parser table as a template for your parser table, this attribute is already set.

2. Define your connector DLL name by adding the following values to the Namespace Class definition in your parser table:

```
KEYWORD AgentPluginDLL
  attribute=eTAgentPluginDLL
  datalocation=agent
  edittype=string
  description=DLL for the Agent plug-in
  default=your_namespace_agent_plug-in_dll
  asciionly=yes
  verbreq=!add
  verbreq=!tomodify
```

your_namespace_agent_plug-in_dll

The DLL that you created for your connector.

More information:

[SASP and the Superagent](#) (see page 89)

Chapter 4: The Sample Connector

The sample connector is used as an example throughout the *Programming Guide for Provisioning*. You should build and compile the sample connector as a prerequisite to developing your own custom connector for the Provisioning Server.

There are several sample connectors included with the Provisioning SDK. You can compile all the sample connectors or a specific connector only. For example, if you plan to work with the ODBC-SDK sample then you can compile only that sample connector.

This section contains the following topics:

[Environment Variables](#) (see page 55)

[Building the Sample SDK Connector](#) (see page 56)

[SDK Connector Code](#) (see page 65)

[Provisioning SDK Directories](#) (see page 65)

[Upgrade Existing Custom Connectors](#) (see page 67)

Environment Variables

Set up your environment before you begin building the sample connector.

The sample connector requires the following environment variable definitions:

Variable Name	Default Installation Directory
<i>PSHOME</i>	\Program Files\CA\Identity Manager\Provisioning Server
<i>PSDK</i>	\Program Files\CA\Identity Manager\Provisioning SDK
<i>PSDKHOME</i>	\Program Files\CA\Identity Manager\Provisioning SDK\admin
<i>%DXHOME%</i>	\Program Files\CA\eTrust Directory\DXserver

Notes:

- *%DXHOME%* is a system environment variable defined when eTrust Directory is installed.
- *PSHOME*, *PSDK*, and *PSDKHOME* are notations of convenience only, which are used throughout this guide.

Building the Sample SDK Connector

Building and integrating the sample SDK connector includes the following major steps:

1. Compile the sample connector.
2. Add the SDK Connector object to the Provisioning Directory.
3. Register the SDK Directory in the sample connector.
4. Explore and correlate the accounts in the SDK Directory.

Compiling the Connector and Generating its Schema

Perform the following procedure to compile the sample connector and generate its schema files.

To compile the sample connector

1. Open a DOS command prompt and change to the following directory:

`PSDKHOME/Samples/SDK`

Note: To verify that your development environment settings are correct, run the `VSvars32.BAT` file before you run `NMAKE` in the next step.

2. Run `NMAKE`, which is called in the three sub-directories (`GUI`, `Agent` and `Populate`). The following files are generated and placed in the `PSDKHOME/bin`, `PSDKHOME/data`, and `PSDKHOME/jiam` directories:

- `CASDKGUI.DLL`
- `SDKPARSE.PTT`
- `SDKNAMESPACE.DLL`
- `SDKPOP.EXE`
- `JIAMEXTSDK.JAR`

Note: `JIAMEXTSDK.JAR` is created only when you set `JIAMJAR=1` in the `GUI/makefile`.

3. Copy the `SDKPARSE.PTT` file from `PSDKHOME/data` to the `PSHOME/data` folder.

Note: This step ensures that `eTrust_sdk.dxc/schema` is copied properly to the `PSHOME/data` folder on a system where the Provisioning Server is installed.

4. Change to the following directory:

`PSDKHOME/Bin`

5. Enter the following command to generate the schema:

```
schemagen -n SDK
```

The following files are generated:

- eTrust_sdk.schema
- eTrust_sdk.dxc

If the SDK is installed stand-alone, the files are placed in *PSDKHOME*/data. If the SDK is installed on the same machine as the Provisioning Server, the files are placed in *PSHOME*/data.

The JIAM Jar File

Java Identity and Access Management (JIAM) is a Java front end to the Provisioning Server.

You can extend the IAM system through the connectors by loading the connector's JAR files.

These JAR files can be built in the connector GUI directory by calling NMAKE, located in the makefile in the *PSDKHOME*/samples/sdk/gui/ subdirectory.

The following is an example that builds the JAR file for the SDK Sample connector:

```
JIAMJAR=1
NAMESPACE=SDK
...
ptts :: $(ADMIN)/Data/sdkparse.ptt
...
! IF "$(JIAMJAR)" == "1"
jars :: $(ADMIN)/jiam/jiamExt$(NAMESPACE).jar
! ENDIF
```

JIAMJAR

Variable that enables or disables the build for the JIAM JAR file. The value 1 means true, and any value other than 1 means false.

NAMESPACE

Name of the connector (SDK, in this case).

You can modify the GUI makefile for your connector to include content similar to that in the example to build the JAR file.

Note: The second block must be put after the "ptts" target because the parser table is required to build the JIAM JAR file.

Generating JAVA Source Files

RDTutility.bat is a tool called by NMAKE to generate the JIAM JAR file. Although the process of generating the JAR file is hidden, you can call the RDTutility.bat directly without using NMAKE. In this way, you have more control over it. For example, you might choose to retain the JAVA source files and the XML files for the connector.

You can specify a config file containing extra information in the PSDKHOME/bin/config to change the default behaviors of the RDT.

To specify a config file to change default behaviors of the RDT

1. Generate Java Objects other than Endpoint, Account, and Policy objects. For example, adding the following lines to the LDAconfig.xml instructs the RDT to generate the LDAP Group Java object as well.

```
<namespace name=LDA">
<objectClass name="LdapGroup" />
</namespace>
```

2. Change the name of attributes used by JIAM to be different from the one defined in the parser table. The following example shows that groupNames are used as the bean property name for the attribute eTLDAMemberOf for LDA connector.

```
<namespace name=LDA">
  <objectClass name="LdapAccount">
    <property>
      <name>groupNames</name>
      <attr>eTLDAMemberOf</attr>
    </property>
  </objectClass>
</namespace>
```

The following example assumes that you have developed a new connector called PKI and that you want to generate the JAVA source files for it.

To generate JAVA source files with the RDTutility.bat batch file

1. Copy the compiled parser table for the connector (should be PKIparse.ptt in this case) into the following directory:

PSDKHOME/data

2. Open the DOS command prompt and change to the directory that you want to use as the base directory to contain the source and XML files.

3. Set the JAVASDK environment variable to point to the JAVA SDK directory by entering the following command:

```
set JAVASDK=C:\j2sdk1.4.2_04
```

Note: This command sets the variable temporarily. If you want to set the variable permanently, make it a system environment variable.

4. Provide extra mapping file (PKIconfig.xml) in the PSDKHOME/bin/config directory.
5. Enter the following command:

```
"PSDKHOME/bin/RDTutility.bat" PKI TRUE
```

PKI

Acronym for the new option.

TRUE

Signifies that the output files are kept.

6. The RDTutility creates a directory called output in the current directory, and places all the JAVA source files and XML files in the following directory structure:

```
/output/PKI/IAM/ObjLayer/
```

Note: The jiamExtPKI.jar file is also generated and copied to the PSDKHOME/jiam directory.

RDTutility.bat Syntax

The following is the syntax for the RDTutility.bat:

```
RDTutility.bat namespaceName keepOutput
```

namespaceName

Acronym for the connector, for example, ADS or N16.

keepOutput

Determines whether to keep the output files. Specify either TRUE or FALSE. The default value is FALSE, if no value is specified.

Extended LDAP JIAM Jar

Extending the Provisioning Server LDA schema requires regenerating the JIAM LDA classes and replacing them in the JIAM.jar. These class files for the LDA connector are in a separate jar called jiamExtLDA.jar, which needs to be added to the classpath of any applications using JIAM. The jiam.jar file includes it in the classpath of its manifest file, so applications have access to it automatically.

Augmenting Self-Service and SPML Service

The SDK files as installed do not contain the files needed to augment self-service and SPML service to understand the SDK connector.

To augment self-service and SPML service

1. Edit `self_service.properties`.
`com.ca.iam.extensions=com.ca.iam.model.options.sdk.impl.SDKOptionDescriptor`
2. Add `jiamExtSDK.jar` to `WEB-INF/lib` in `iamwebi_self_service.war` and `iamspml.war`.
3. Restart Tomcat.
4. In SPML Service Configuration, under Additional Namespaces, add the following:
`com.ca.iam.model.options.sdk.impl.SDKOptionDescriptor`

Compiling SDK Samples on Solaris

To create and compile from a Provisioning SDK build and development environment on Solaris, complete the following steps.

To compile SDK samples on Solaris

1. Install Sun Studio 10 or 11.
2. Install `make-3.81` (from www.sunfreeware.com).
3. Install `gcc-3.4.6-sol-sparc-local` (from www.sunfreeware.com).
4. Install `libiconv` (from www.sunfreeware.com).
5. Set the following environment variables: The `setetasdkenv.sh` script sets `DEVROOT` and `LD_LIBRARY_PATH`.
6. Add the Sun Studio bin directory to the `PATH` (for example, `/opt/SUNWspro/bin`).
7. Add `/usr/local/bin` to in the `PATH` (GNU make is installed in this directory by default).
8. Change to the SDK directories under this path, where `<module>` is the component to be compiled:
`/opt/CA/ProvisioningSDK/admin/samples/<module>`

9. Run the following from the command line:

```
make -f makefile.unix
```

The generated libraries are placed in the following directory:

```
/opt/CA/ProvisioningSDK/admin/lib
```

Adding the SDK Connector to the Provisioning Directory

To add the namespace (connector) object to the Provisioning Directory, perform the following procedure on the machine running the Provisioning Server.

To add the Connector object

1. Double-click the Services icon in the Control Panel.
The Services window appears.
2. Stop the Provisioning Server and the superagent.
3. Click Close.
4. Close and exit from your current Provisioning Manager session.
5. Copy the following files from *PSDKHOME/bin* to *PSHOME/bin*:
 - CASDKGUI.DLL
 - SDKPOP.EXE
 - SDKNAMESPACE.DLL
6. Copy the SDKParse.ptt file from *PSDKHOME/data* to *PSHOME/data*.
7. If SDK is not installed on the same machine as the Provisioning Server, copy the eTrust_sdk.schema from *PSDKHOME/data* to *PSHOME/data*.
8. Copy the eTrust_sdk.dxc from *PSDKHOME/data* (or *PSHOME/data* if the SDK is installed on the same machine as the Provisioning Server) to *%DXHOME%/config/schema*.
9. Create the *PSHOME/SDK* directory.
10. Copy the *PSDK/SDK_Directory* and all subfolders into the *PSHOME/SDK* directory.
11. Reinitialize the eTrust Directory etrustadmin Service by entering the following command:

```
dxserver init etrustadmin
```
12. Restart the Provisioning Server and the superagent.
13. Open a DOS session and change to the following directory:

```
PSHOME/bin
```

14. Enter the following command:

```
sdkpop <domain_name> <administrator_name>  
<administrator_password>
```

15. Register the sample directory.

More information:

[Registering the Sample Directory](#) (see page 63)

Note: Typically, you edit the *PSHOME*/data/etrust_admin.conf file to include the schema file for your connector.

For Windows, you do not need to add the SDK schema file to etrust_admin.conf because it is already included.

For UNIX, you must perform the following:

1. Log in as the etaslapd user.
2. Copy the etrust_sdk.schema file from the SDK data folder to the Provisioning server data folder:

```
cp /opt/CA/eTrustAdminSDK/data/etrust_sdk.schema  
/opt/CA/eTrustAdminServer/data/etrust_sdk.schema
```

3. Edit the .../data/etrust_admin.conf file in the Provisioning server:

```
include "/opt/CA/eTrustAdminServer/data/etrust_admin.schema"  
include "/opt/CA/eTrustAdminServer/data/etrust_dyn.schema"  
#include "/opt/CA/eTrustAdminServer/data/etrust_sdk.schema"  
uncommenting the ".../etrust_sdk.schema" line.
```

How the Provisioning Server Handles a Connector Object Addition

The following sequence of events occurs when you add a namespace (connector) object:

1. The Provisioning Server server starts, and its back end reads the SDKPARSE.PTT parser table and loads the saspRegister.DLL.

The parser table indicates to the Provisioning Server back end the DLL to be loaded. In this example, it is saspRegister.DLL.

2. The SDKPOP.EXE program creates the following three objects in the Provisioning Directory:
 - SDK Namespace
 - SDK Policies
 - SDK DefaultPolicy
3. When Provisioning Server restarts, the Provisioning Server back end provides Manager with the SDK objects in the Provisioning Directory.

Registering the Sample Directory

Perform the following procedure to register the sample directory.

To register the sample directory in the Provisioning Manager

1. Select the Namespaces task frame.
2. Select the SDK Directory object type.
3. Click New.
4. Enter the following values in the SDK Directory dialog:
 - *Directory Name - sdksample
 - *Host Name - Provisioning Server host name
 - *IP Address - Provisioning Server IP address
 - Policy - "SDKDefaultPolicy"
 - Comments - "SDK Test Machine"
5. Click OK.

How the Provisioning Server Handles the Directory Registration

The following sequence of events occurs when you register the SDK directory:

1. The GUI plug-in that controls the SDK Directory dialog sends an ADD request using the information in the dialog to the Provisioning Server.
2. The Provisioning Server sends the ADD request and information to its back end.
3. The back end parses the ADD request and determines which server plug-in handles the request. In this sample, the Superagent Server Plug-in (SASP) handles it.
4. SASP binds to the superagent back end and sends two ADD requests to add the following:
 - SDK Connector object
 - SDK Directory object

5. The superagent back end responds to the first ADD request and loads the DLL for the SDK Connector object.
6. The superagent back end responds to the second ADD request and sends the ADD request to the SDK Connector agent plug-in.
7. The SDK Connector agent plug-in locates the SDK directory and sends results back to SASP.
8. SASP adds the SDK directory object to the Provisioning Directory and then sends the results back to the GUI plug-in for display.

Exploring and Correlating SDK Directory Accounts

Perform the following procedure to explore and correlate the accounts in the SDK directory.

To explore and correlate the accounts in the SDK directory

1. Select the SDK Directory object in the list view.
2. Click your right mouse button and choose Explore/Correlate.
3. Complete the Explore and Correlate Directory dialog.
4. Click Start.

How the Provisioning Server Handles the Directory Exploration and Correlation

The following sequence of events occurs when you explore and correlate the directory:

1. The GUI plug-in that controls the Explore and Correlate Directory dialog sends an Explore request for all objects in the directory to the Provisioning Server. The Explore request is a special LDAP Search request asking for the eTExploreUpdateEtrust attribute.
2. The Provisioning Server sends the SEARCH request to its back end.
3. The back end parses the SEARCH request and determines which server plug-in handles the request. In this sample, SASP handles the request.
4. SASP binds to the superagent back end and sends a sequence of Search requests to the agent plug-in that is in charge of the SDK Connector. Separate Search requests are made for the account container, the account objects, the group container, and the group objects.
5. The SDK Connector agent plug-in accesses the SDK directory and sends results back to SASP.
6. SASP adds the accounts, groups, containers, and any other objects specified in the parser table to the Provisioning Directory.
7. SASP sends exploration statistics back to the GUI plug-in for display.

8. The GUI plug-in sends a Correlate request to the Provisioning Server. The Correlate request is a special LDAP Search request asking for the eTEExploreCorrelateUsers attribute, and optionally, the eTEExploreCreateUsers attribute.
9. SASP creates a global user object (or locates an existing global user) for each account it finds, and creates an inclusion that links the account and global user objects.
10. SASP sends correlation statistics back to the GUI plug-in for display.

SDK Connector Code

The code for the SDK Connector includes a GUI plug-in and an agent plug-in. The complete SDK Connector code sample is located in the following directory:

`PSDKHOME/Samples/SDK`

The GUI plug-in code is located in the GUI subdirectory:

`PSDKHOME/Samples/SDK/GUI`

The agent plug-in code is located in the Agent subdirectory:

`PSDKHOME/Samples/SDK/Agent`

Note: The SDK provides an additional code sample that you can use when building an agent plug-in with a dynamic hierarchical structure. The source for this sample can be found in the following directory:

`PSDKHOME/Samples/SDK/SuperAgent/SampleDDIT`

Provisioning SDK Directories

The following table lists the Provisioning SDK directories:

Directory	Description
<code>PSDK/opt/src/ecs</code>	Provisioning Server common utilities
<code>PSDK/opt/lib</code> <code>PSDK/opt/include</code>	OpenLDAP library and include files for use by client code
<code>PSDKHOME/bin</code>	Generated binaries
<code>PSDKHOME/common/unicomn</code>	Common services header files
<code>PSDKHOME/data</code>	Compiled parser tables

Directory	Description
PSDKHOME/include	Provisioning Server header files
PSDKHOME/include/GUI	GUI plug-in header files
PSDKHOME/include/PTI	Parser table include files (base class definitions)
PSDKHOME/lib	Generated libraries (in debug mode)
PSDKHOME/samples/SDK	Sample code provided with the SDK

Refresh the Visual C++ Compilation Environment

If you are using a Windows 2000 or Windows XP machine, compilation errors may occur when compiling the SDK samples. An example of such an error follows.

```
===NMAKE Build Started...
  IFR c Compiling, ./Debug/CASDKGUI.obj from ./CASDKGUI.cpp
CASDKGUI.cpp
./CASDKGUI.cpp(20) : fatal error C1001: INTERNAL COMPILER ERROR
      (compiler file 'mscl.cpp', line 1794)
      Please choose the Technical Support command on the Visual C++
      Help menu, or open the Technical Support help file for more
information
      NMAKE : fatal error U1077: 'cl' : return code '0x2'
      Stop.

      D:/Program Files/CA/eTrust/Admin/SDK/eTrust/Admin/Samples/SDK/GUI>set
makeerrorcode=2
      Building of "D:/Program
Files/CA/eTrust/Admin/SDK/eTrust/Admin/Samples/SDK/GUI" +++++ FAILED +++++
      NMAKE : fatal error U1077: 'IF' : return code '0x2'
      Stop.
```

To fix this error, you need to refresh the Visual C++ compilation environment.

To refresh the Visual C++ compilation environment

1. Open the Control Panel, System dialog.
2. Click the Advanced tab.
3. Open the Environment Variables dialog.
4. Select the Path variable in the System Variables list.
5. Edit the Path variable.
6. Verify the path variable by clicking OK.
7. Click OK to validate the transaction.
8. Open a new command prompt and recompile the SDK samples.

Upgrade Existing Custom Connectors

If you have built custom connectors to handle data in your environment, you must recompile these connectors against the latest SDK to make them compatible with the latest release of the Provisioning Server.

Note: You must install the Provisioning Server or a subsequent release before performing this process.

To upgrade existing custom connectors

1. Compile the custom connectors.
2. Install the parser table file (*.ptt). Add the SDK Namespace Object.
3. Run the SCHEMAGEN utility.

More information:

[Generating Schema Files](#) (see page 37)

4. Install the connector Schema file in PSHOME/Data.
5. Install the connector DXC file in %DXHOME%/config/schema.

Note: This procedure lets the custom connector run with the Provisioning Server.

The Provisioning Server supports multi-threaded agent plug-ins (connectors). If you want to make your connectors multi-threaded, you must make code changes. The most important thing is to make your agent plug-in thread-safe and multi-threaded, and register it as thread-safe and multi-threaded with the Provisioning Server.

More information:

[Enable Thread Support of Agent Plug-ins](#) (see page 168)

Chapter 5: Predefined Object Classes

The most important Provisioning Server object classes and objects are already defined for you. These predefined classes contain the organizational structure and behaviors that objects must have to work in the Provisioning Server. The classes that you define for your own endpoint types are subclasses of three predefined classes.

This section contains the following topics:

[Base Classes](#) (see page 69)

[Common Object Classes](#) (see page 70)

[Inclusion Objects](#) (see page 73)

Base Classes

Base classes organize endpoint type objects in the Provisioning Server and provide built-in behaviors that the objects must have. The base classes in the Server Framework are:

- eTAccount
- eTDirectory
- eTPolicy
- eTContainer

Define a Subclass to a Base Class

Objects in base classes are not instantiated. Therefore, account, directory, policy, and container objects are defined in the parser table as subclasses to the base classes.

To define a subclass to a base class

1. List the base class as the value for the SUPERCLASS property.
2. Use the #include statement in your parser table to pull in common attributes defined by PTI include file for the base class.

Common Object Classes

Common objects record relationships between objects. In the Provisioning Server, the following are the predefined areas for common object classes:

- Namespace
- Directory
- Account
- Policy
- Global User
- Program Exit
- Inclusion

Namespace Class

eTNamespace

This object class is the top-level node under which all endpoint type objects, except account template containers, are placed. Before you add an endpoint type, the Provisioning Server places all objects under a predefined endpoint type called Common Objects. The Common Objects endpoint type is used to:

- Define a location for objects
- Write behaviors for the objects using the same extensible plug-in mechanism that is available for real connector objects
- Display the account template container objects

Directory Class

eTDirectory

This object class is the base class for each eTXXXXDirectory class of an endpoint type. It defines common attributes for all directories (endpoints).

eTContainer

This object class is used in hierarchical endpoint types as the base class for each container class that can hold account objects.

Account Class

eTAccount

This object class is the base class for each eXXXXXAccount class in a endpoint type. This object class defines common attributes for all accounts.

Policy Class

eTPolicy

This object class is the base class for each eXXXXXPolicy class in a endpoint type. It defines common attributes for all account templates. An account template object is similar to an account object because it is used as the template for creating new accounts. An account template object is stored in the Provisioning Directory. Dragging and dropping a global user on an object that is a subclass of eXXXXXPolicy creates an account for the global user in each directory associated with the template, using values computed from rules contained in the account template.

eXXXXXPolicyContainer

This object class is a container where account templates of the XXXX endpoint type are created in the Common Objects connector.

eTRole

This object class represents a provisioning role--a group of eXXXXXPolicy objects for different endpoint types. Assigning a provisioning role to a global user (with user synchronization) updates the global user's eTRoleDN attribute and creates accounts for the global user in each endpoint associated with each account template in the provisioning role.

eTRoleContainer

This object class is a container where provisioning roles are created in the Common Objects endpoint type.

Global User Class

eTGlobalUser

This object class is a global user object. Global user objects link users to their accounts.

eTGlobalUserContainer

This object class is a container where global users are created in the Common Objects endpoint type.

eTGlobalGroup

This object class represents a group of global user objects.

eTGlobalGroupContainer

This object class is a container where global groups are created in the Common Objects endpoint type.

eTAdminProfile

This object class represents an admin profile. An admin profile is a collection of privilege strings describing administrative actions.

When using a client of the Provisioning Server such as the Provisioning Manager, assigning an admin profile to a global user by setting the global user's eTUserAdminProfile attribute gives that global user the privileges to carry out administrative actions in the Provisioning Server.

eTAdminProfileContainer

This object class is a container where admin profiles are created in the Common Objects endpoint type.

Program Exit Class

eTExit

This object class is a program exit object. It is used for common exits and native exits. For common exits, the eTSubclass attribute is unset. For native exits, the eTSubclass value is "XXXXExit".

eTExitContainer

This object class is a container where program exits are created. Common exits are placed in this container in the Common Objects endpoint type. Native exits are placed in this container in the respective endpoint type.

Inclusion Class

eTInclusionObject

This object class forms a relationship between two objects in the Provisioning Server. An inclusion object can be placed only under an eTInclusionSubordinate container.

eTInclusionSubordinate

This object class is a container partitioning inclusion objects according to the subordinate or child object class of the inclusion relationship. One of these containers can be placed only under an eTInclusionSuperior container.

eTInclusionSuperior

This object class is a container partitioning inclusion objects according to the superior or parent object class of the inclusion relationship. One of these containers can be placed only under the eTInclusionContainer container.

eTInclusionContainer

This object class is the top-level container under which all inclusion objects and related containers are placed. This container is placed under the Common Objects endpoint type container.

Inclusion Objects

Inclusions represent relationships between objects. The Provisioning Server uses the eTInclusionObject object class to represent inclusions. You must define which inclusions are permitted in your parser table. You do this by setting properties to enable the creation of inclusions between your object classes. The GUI framework will only permit drag or drop actions between object classes that permit inclusions.

The Provisioning SDK provides several predefined inclusion relationships, required inclusion relationships, and optional inclusion relationships that you can define. For example:

- You do not have to define inclusion relationships between common objects, such as between a global user and a global user group.
- You must define inclusion relationships between certain connector objects, such as between an endpoint and an account template.

- You must define inclusion relationships between connector objects and common objects, such as between a global user and an endpoint.
- You can create optional inclusion relationships between connector objects that existed before you explored the directory, such as between a Windows NT account and a Windows NT group.

Predefined Inclusions

Common object inclusion relationships are already defined for you. You do not need to add any attributes to your parser table to creation the inclusion. For example, you do not have to define an inclusion between a global user group and a global user.

Required Inclusions

You must add the Parent_Class or Child_Class properties in your parser table to enable creation of inclusions between objects of the following object classes:

- eTGlobalUser (parent) and eTXXXXAccount (child)
- eTXXXXPolicy (parent) and eTXXXXDirectory (child)
- eTRole (parent) and eTXXXXPolicy (child)

In addition, you must define additional inclusion relationships between objects, even though the Provisioning Server does not represent these relationships using inclusion objects stored in the Provisioning Directory, as follows:

- eTXXXXDirectory (parent) and eTGlobalUser (child)
- eTXXXXDirectory (parent) and eTGlobalGroup (child)
- eTXXXXPolicy (parent) and eTGlobalUser (child)
- eTXXXXPolicy (parent) and eTGlobalGroup (child)
- eTXXXXPolicy (parent) and eTXXXXAccount (child)

Attempts by the GUI plug-in (or any other LDAP client) to create any of these special inclusion relationships will result in some other action being carried out by the Provisioning Server:

More information:

[Required Inclusions - Example 1](#) (see page 75)

[Required Inclusions - Example 2](#) (see page 75)

Required Inclusions - Example 1

When you drag and drop an account onto a global user, an LDAP Add request is sent to the Provisioning Server. This is a request to an inclusion between the account and the global user, thus correlating the account to the global user. The Provisioning Server creates this inclusion object in this provisioning directory. Furthermore, this specific inclusion is such that each account can be correlated to at most one global user, so this request first deletes any prior inclusion object linking this account to a global first.

Required Inclusions - Example 2

When you drag and drop a Provisioning Role onto a Global User, an LDAP Add request is sent to the Provisioning Server. This is a request to an inclusion between the global user and the provisioning role, thus assigning the user to the role and creating or updating any applicable accounts. Since role assignments are not maintained as inclusion objects in the provisioning directory, the provisioning server turns this inclusion Add request into a modification of the eTRoleDN attribute of the global user and then performs the applicable user synchronization function to create or update accounts.

Optional Relationships

The Provisioning Server does not use inclusion objects to represent pre-existing relationships between connector objects, such as between an account and group. When defining the schema for your objects, there are two ways to show that two objects are related:

- The DIT hierarchy
- Multi-valued attributes

By placing an object in the DIT hierarchy under another object, you can make the relationship between the objects explicit. However, doing so makes it awkward to change the relationship because this changes the DN of one of the objects. Also, using the DIT hierarchy limits the number of objects that can be related to another, since each object being managed should have only one DN.

A better solution for handling the group membership is to define a multi-valued attribute in the account object, the group object or both objects. The values in the multi-valued attribute depend on the type of connector that you manage:

- Non-hierarchical endpoint types use one container for each leaf object. Because of this, the value for each multi-valued attribute can be the simple name of the group or account.

For example, the path name of an account is the following:

```
eTSDKAccountName=John  
Doe,eTSDKAccountContainerName=Accounts,eTSDKDirectoryName=SDK Directory,  
eTNamespaceName=SDK Namespace,dc=domain_name,dc=eta
```

The value of the multi-valued attribute is the following:

John Doe

- Hierarchical endpoint types use path names to identify the account or group. If you are identifying an account, use a partial DN relative to the top of the connector directory as the value of the multi-valued attribute.

For example, the path name of an account is the following:

```
eTSDKAccountName=John Doe,eTSDKOrganizationalUnitName=System  
Accounts,eTSDKDirectoryName=SDK Directory, eTNamespaceName=SDK  
Namespace,dc=domain_name,dc=eta
```

The value of the multi-valued attribute is the following:

```
eTSDKAccountName=John Doe,eTSDKOrganizationalUnitName=System Accounts
```

To manage the group memberships from your GUI plug-in, you can perform direct manipulation of the multi-valued attribute, or you can pretend that the relationship is really represented with inclusion objects. If you choose to make your GUI plug-in manipulate the multi-valued attributes directly, the following can occur:

- The drag-and-drop feature becomes unavailable.
- The GUI plug-in is unable to use the standard `CosInclusionPage` class for managing group memberships.

To pretend that the relationship is represented with inclusion objects, define the following:

- `CHILD_CLASS` or `PARENT_CLASS` class-level properties in the parser table
- `INCLUSIONPARENT` or `INCLUSIONCHILD` attribute-level property in your multi-valued attribute.

The client library routines that add, delete, and search for inclusion objects will be aware of the `INCLUSIONPARENT` or `INCLUSIONCHILD` settings, and access or modify the multi-value attribute instead of sending inclusion object requests to the Provisioning Server.

Inclusion Sample Code

Required Inclusion

To permit the creation of the standard inclusions, your parser table must define the following Parent_Class, Child_Class, and Draggable property settings:

```
CLASS XXXXDirectory,eTXXXDirectory
    superclass=eTDirectory
    child_class=eTGlobalUser
    child_class=eTGlobalGroup
    parent_class=eTXXXPolicy
    draggable=yes
CLASS XXXXAccount,eTXXXAccount
    superclass=eTAccount
    parent_class=eTGlobalUser
    draggable=yes
CLASS XXXXPolicy,eTXXXPolicy
    superclass=eTPolicy
    parent_class=eTRole
    child_class=eTGlobalUser
    child_class=eTGlobalGroup
    child_class=eTXXXAccount
    draggable=yes
```

Optional Inclusion

To permit the manipulation of a connector-specific relationship as though it were represented using inclusion objects, define the multi-valued attribute in the parent or child class. The following example shows an inclusion between a group and an account using a multi-valued attribute in the parent class:

```
CLASS XXXXGroup,eTXXXGroup
  child_class=eTXXXAccount
  draggable=yes
KEYWORD MemberAccount
  attribute=eTXXXMemberAccount
  multivalued=yes
  inclusionChild=eTXXXAccount
CLASS XXXXAccount,eTXXXAccount
  draggable=yes
```

To represent container classes in a hierarchical connector, add the following definitions.

```
CLASS XXXXOrganizationalUnit, eTXXXOrganizationalUnit
  superclass=eTContainer
  child_class=eTGlobalUser
  child_class=eTGlobalGroup
  draggable=yes
```

Chapter 6: Endpoint Acquisition

The Provisioning Server can perform an acquisition of all the endpoint systems in your enterprise. An acquisition does the following:

- Registers the endpoint, gathering the necessary connection information for the endpoint and creating the endpoint object in the Provisioning Directory.
- Explores accounts and other objects and populates them in the Provisioning Directory, permitting them to be displayed in Manager and other client applications.
- (Optionally) Performs additional actions against the accounts now residing in the Provisioning Directory:
 - Correlates the accounts with existing global users
 - Creates users, correlating the accounts with global users created to have the same names as the accounts
 - Updates users with values of selected account attributes to update attributes of global users previously correlated to the accounts

Note: This chapter includes instructions for using Provisioning Manager to explore, correlate, and update users. You can also use the Identity Manager User Console for these tasks as explained in the *Provisioning Guide*.

This section contains the following topics:

[How to Register the Endpoint](#) (see page 79)

[How to Explore the Endpoint](#) (see page 80)

[How to Correlate the Endpoint](#) (see page 84)

[How to Update Users](#) (see page 86)

How to Register the Endpoint

The registration process begins with the Provisioning Manager.

1. An administrator uses the Endpoints task, selects the endpoint object class for the Namespace of the directory to acquire, and clicks New. This action sends a New Directory command to the GUI plug-in.
2. The GUI plug-in receives the command and displays a Directory dialog that collects the connection information for an existing, but unmanaged, endpoint.

Note: The GUI plug-in can obtain the endpoint name and connection information from an external source.

3. The registration process begins when the administrator clicks OK. This action sends an LDAP Add request to the Provisioning Server requesting that the eTXXXXDirectory object be added under the eTNamespace node for this connector in the current Provisioning Server domain:

```
eTXXXXDirectoryName=DIRNAME,  
eTNamespaceName=NS-NAME,  
dc=DOMAIN,  
dc=eta
```

4. The Provisioning Server back end receives the LDAP Add request, and sends it to the server plug-in responsible for the eTXXXXDirectory object class.
5. The server plug-in sends an LDAP Add for this endpoint object to a connector running within a C++ or Java connector server. The connector verifies the connection information and, if it can connect to the endpoint, makes the endpoint object available to the connector server DIT.

Note: If this is the first endpoint added to the Connector Server since the Connector Server started, then the server plug-in also sends an LDAP Add for the endpoint type object to the Connector Server, to instruct the Connector Server to load the connector.
6. The server plug-in also sends an LDAP ADD request to the Provisioning Directory to add the endpoint object there. From this point forward, the Provisioning Server recognizes the object as a managed endpoint object with all connection parameters recorded.

How to Explore the Endpoint

The exploration begins after the endpoint is registered.

1. An administrator right-clicks the endpoint object, and selects Explore/Correlate from the pop-up menu.

This action sends an Explore and Correlate Directory command to the GUI plug-in.

2. The GUI plug-in receives the command and displays an Explore and Correlate Directory dialog that shows the following:
 - The container tree for the endpoint. You specify which portion of the endpoint is explored by checking the entire endpoint or specific containers in the endpoint (with either one-level scope or full sub-tree scope).
 - A series of checkboxes and radio buttons from which you select actions to perform on the selected containers.

3. The exploration process begins when you click Start.

This action sends one or more LDAP SEARCH requests to the Provisioning Server to cover the scope of containers that you checked in the container tree. These requests specify the base DN, either the endpoint or container DN, and specify sub-tree or one-level scope arguments depending on how the endpoint or container node is checked.

In the simplest case, you would check the endpoint node with a sub-tree scope, resulting in a single LDAP search with sub-tree scope and with the endpoint DN as the search base DN.

These requests are identified as explore and not normal search requests because of the presence of the eTEExploreUpdateEtrust attribute in the list of requested attributes that LDAP lets the client specify.

Similarly, you can substitute different attribute names for eTEExploreUpdateEtrust to request the three other special actions (Correlate, Create Users, Update Users).

If you were to initiate explore requests from ETAUTIL or some other LDAP client application, you must form search requests specifying the endpoint or container node and scope arguments.

The following additional options are available when you use these interfaces:

Set the Search Scope Parameter to Base

To explore a specific named object, set the search scope parameter to "base" instead of sub-tree and one-level. This can be used to add, delete, or modify a single account or other object so that what is stored in the Provisioning Directory matches the state of that account or other object in the managed endpoint.

Include eTEExploreReportAdditions, eTEExploreReportDeletions, or both with eTEExploreUpdateEtrust in the List of Requested Attributes

The Provisioning Server will report back the names of objects that are being added or removed from the Provisioning Directory as part of this explore operation.

Include eTEExploreReportAdditions, eTEExploreReportDeletions, or both *without* eTEExploreUpdateEtrust in the List of Requested Attributes

The Provisioning Server will report back a list of accounts that would have been added or removed (but not actually update the Provisioning Directory).

Use Filter "objectclass=*" for all Requests Sent from Manager

Restrict the set of objects to those that match a pattern.

Filter "objectclass=*" explores all objects in the requested scope. For example:

```
"(&(objectclass=eTXXXXAccount)(eTXXXXAccountName=a*))"
```

restricts the explore to consider only accounts and only those accounts whose name begins with "a". All objects not matching the filter are ignored and are neither added, deleted, nor modified in the Provisioning Directory in this explore request.

When the Provisioning Server receives the explore request, it breaks the scope of the explore request down so that it processes one object class and one container at a time. By performing one-level (or base) searches, the Provisioning Server obtains lists of objects both from the Agent and from the Provisioning Directory.

Important! If some of your endpoint objects are not found during an exploration, check the DITPARENTCLASS and DITCHILDCLASS properties in containers and leaf nodes. If the Provisioning Server does not determine that a particular object class can appear in a particular container, it does not search for that object class.

The request sent to the agent requests attributes that need to be stored in the Provisioning Directory. For non-account object classes, this includes attributes whose parser table definition includes DATALOCATION=both. If no attributes are identified with DATALOCATION=both, the object should not be stored in the Provisioning Directory in the first place, and is skipped. This would be the case for a resource, process, or other highly volatile object when you decide it is better to not explore the object, but instead rely on communicating with the endpoint whenever any client needs to receive a list of these objects.

For account object classes, additional attributes are retrieved and stored in the Provisioning Directory beyond those identified with DATALOCATION=both. These are any attributes that have been configured as relevant for any of the later phases of explore (Correlate, Create Users, or Update Users). By storing these extra account attributes in the Provisioning Directory, the Provisioning Server can more easily perform the Correlate, Create Users, or Update Users actions without further accessing the endpoint.

Using the object lists returned from the Provisioning Directory and agent plug-in, the Provisioning Server identifies and performs additions, deletions and updates of the objects stored in the Provisioning Directory.

Note: If your connector has a dynamic hierarchy, the management of containers in the endpoint involves some additional actions. A dynamic hierarchy is one that is not fixed for all endpoints of your connector; but instead uses organization, organizational unit, or other such container classes to organize objects to installation-specific groupings. When you have a dynamic hierarchy, you might want the administrator to be able to designate that the Provisioning Server is managing only a portion of the entire hierarchy. Your responsibility then falls into the following areas:

- Identifying that you have a dynamic hierarchy.
- Providing support in your GUI and Agent plug-ins to see containers in the Container Tree of Explore/Correlate dialog before they have been explored.
- Controlling when containers are populated into the Provisioning Directory, so they can be seen in the Content dialog.

You can identify a container object class as dynamic in the parser table by setting the following class-level properties:

- dynamiccontainer=yes
- predefined=no
- hidden=no

In your agent plug-in, set the DYNAMIC_DIT mode bit on the parent container of the dynamic container.

You manage the container tree by providing the function ChildrenToExplore() in your GUI plug-in. The default behavior of this function displays only those containers that have previously been explored. If you do not provide the ChildrenToExplore () function, the administrator can only manage the entire endpoint. In your ChildrenToExplore () function, you are asked to provide a complete list of containers at each level as the administrator expands various containers in the Container Tree.

This function cannot be implemented as a normal one-level search against the supplied endpoint or container node, as a one-level search is typically satisfied by reading the Provisioning Directory only. Instead, the ChildrenToExplore() function is implemented as a one-level search including the special attribute eAgentOnly in the list of requested attributes. The eAgentOnly attribute is a control attribute that tells the provisioning server to send the search operation to the connector instead of to the provisioning directory. Thus this search returns the containers present on the endpoint as opposed to the containers previously explored.

Finally, you control when containers are populated into the Provisioning Directory by sending special LDAP ADD and LDAP DELETE requests to the Provisioning Server. These requests are sent automatically by the GUI framework, based on the information in the parser table above, but you need to understand what is happening.

Dynamic Containers

Dynamic containers are expected to be added to the Provisioning Directory only when they are explicitly explored. A one-level explore of one container is interpreted as a request to add leaf nodes (such as accounts and groups) and static sub-containers, but not dynamic sub-containers. If you manage only one level under the organizational unit of "North America," child organizational units such as "California" or "New York" are not yet managed. Therefore, do not populate these organizational units into the Provisioning Directory, which causes them to appear in the Content dialog for the directory.

Note: During one-level explores, the Provisioning Server ignores dynamic sub-containers.

Dynamic containers are added only when a client sends a special LDAP ADD request, which sets `eExploreUpdateEtrust` equal to 1. This sends the ADD request to the agent plug-in, which confirms that the container exists, and adds the container to the Provisioning Directory. If the `eExploreUpdateEtrust` attribute is not included in the ADD request, the Provisioning Server and Agent plug-in assume this is a request to add a new container, which would fail with an "already exists" error. As you navigate the container tree in the Explore/Correlate dialog, the GUI framework sends a special explore LDAP ADD requests for each dynamic container it expands and LDAP DELETE requests to "unexplore" the added containers if you leave the dialog without exploring any sub-container added. Also, for each container that is explored with a sub-tree or one-level scope, the GUI framework sends a special explore LDAP ADD request for the container preceding the special explore LDAP SEARCH to explore both the container and its content.

How to Correlate the Endpoint

Correlation begins after the endpoint is registered and explored. The following steps are involved:

1. Open the Explore and Correlate dialog for the endpoint, and perform the following:
 - a. Select one or more nodes in the container tree on which to operate.
 - b. Select the "Correlate Accounts to Global Users" option, with either the Use Existing Global Users or Create Global Users as Needed option.
 - c. Click Start.

This action sends one or more special correlate LDAP SEARCH commands to the Provisioning Server. These correlate commands are sent by the GUI framework. The following description tells you what is happening, but no coding is required on your part.

As with the special explore LDAP SEARCH commands, the special correlate LDAP SEARCH command is identified as such by the presence of a special attribute name in the requested attribute list of the SEARCH command. If the administrator selects the "Use Existing Global Users" option, then the special attribute requested is eTExploreCorrelateUsers. However, if the administrator selects the "Create Global Users as Needed" option, then the special attribute requested is eTExploreCreateUsers.

2. The Provisioning Server back end receives the special correlate LDAP Search request and performs the Correlate or Create Users behavior. This involves the following steps:
 - a. Determining on which accounts to operate.
 - b. Skipping any account that is already correlated to a global user.
 - c. Performing the Correlate or Create User behavior for each account.

The Provisioning Server checks the LDAP Search parameters (base DN, scope, and filter) and parser table information to determine the accounts on which to operate. The Provisioning Server sends base or one-level LDAP SEARCH requests to the Provisioning Directory, and the returned account list (and account attributes) is used to drive the Correlate or Create Users operation.

Any account already correlated is skipped. This involves checking to see whether an inclusion object exists that links this account to any global user (even to [default user]). An account already correlated is not processed again. You must use the "Remove Account From User" menu item to remove the existing inclusion object in order for subsequent Correlate or Create User operations to reprocess the account.

3. The accounts that are not already correlated are processed according to the specifics of Correlate or Create Users, respectively.
 - Correlation with the Use Existing Global Users option attempts to match the account with one of the existing global users. It does so by sending LDAP Search requests to the Provisioning Directory using search filters that check global user attributes against account values read from the Provisioning Directory. The attributes compared are controlled by the customer by using the Provisioning Manager. Since the account attribute values that are used are read from the Provisioning Directory, there is a slight risk of using stale information. This could happen if the attribute used in matching was recently modified by a tool other than the Provisioning Server and the Correlate or Create Users request was issued without a recent explore request. In such cases, the Provisioning Server does not know that the attribute value has been updated.

If a single matching global user is found, an inclusion object is created to link the account and the global user. Otherwise, an inclusion object is created to link the account and the [default user] global user.

- Creating users with the Create Global Users option attempts to create a new global user for each account. No attempt is made to see if the account matches an existing global user. Instead, a global user entry with eTGlobalUserName, eTUserid, and eTUID attributes computed according to parser-table defined mapping is built up and sent in an LDAP ADD request to the Provisioning Directory. Parser table mappings refers to the use of the "globaluserattribute" property settings on account attributes. If not explicitly set with a "globaluserattribute" property, the account attribute that maps to both eTGlobalUserName and eTUserid is the account's naming attribute.

After the global user is created (and even if the creation reports an "already exists" error), the inclusion object that links the account to the global user is created.

How to Update Users

Run the update users phase after the accounts have been correlated. The steps are the following:

1. Open the Explore and Correlate dialog for the endpoint, and perform the following:
 - a. Select the nodes in the container tree that you want to explore or correlate.
 - b. Select the "Update Global Users" option.
 - c. Click Start.

This action sends one or more special update-users LDAP SEARCH commands to the Provisioning Server. These updates are sent by the GUI framework. The following describes the process, but no coding is required on your part.

As with the special explore and correlate LDAP SEARCH commands, the special update is identified by the special attribute name in the requested attribute list of the SEARCH command. In this case, the special attribute requested is eTExploreUpdateUsers.

2. The Provisioning Server receives the special update and performs the Update Users behavior. This involves the following steps:
 - a. Determining which global user attributes to update from account attributes; if none, aborts the entire Update Users operation.
 - b. Determining which accounts to operate on.
 - c. Skipping or failing any update from an account that is not yet correlated to a global user, or any global user correlated to a "restricted" global user.
 - d. Performing the Update User behavior for each account.

The set of attributes to update is controlled by the `eUserUpdateMap`, `eCustomUserUpdateMap`, and `eDefaultUserUpdateMap` attributes in the endpoint object. If `eCustomUserUpdateMap` is "1", then the mapping is defined by the `eUserUpdateMap` attribute. Otherwise, the value seen in the virtual `eDefaultUserUpdateMap` attribute is used. The default mapping is computed by the Provisioning Server as the value taken from the `eUserUpdateMap`, `eCustomUserUpdateMap`, and `eDefaultUserUpdateMap` attributes of the connector node.

The default mapping at the connector level is computed using parser table-defined mappings ("globaluserattribute" properties set on account attributes) as updated to include any custom correlation matching relationships defined through Domain Properties in the Provisioning Manager.

The set of accounts on which to operate is determined in the same manner as correlation operations.

Accounts not correlated to a global user are skipped, since there would be no global user to update. Restricted global users are global users such as the [default user] who do not represent a person possessing attributes that derive from account attributes; these accounts are not updated.

For accounts that are correlated to a global user, the global user entry is read from the Provisioning Directory, and attributes are updated, as necessary, to correspond to the desired attribute values taken from the account attributes. As with the Correlate and Create Users operations, the account attributes used are those updated by the most recent explore of the account. If no recent explore has been performed and administrators use tools other than the Provisioning Server to update these account attributes, the global users may be updated with stale attribute values.

Chapter 7: Superagent Server Plug-in

The Server framework implements server behavior that is common across all connectors using server plug-ins. Server plug-ins are runtime loadable DLLs that do the following:

- Accept and validate requests from client interfaces
- Search, read, and store information in the Provisioning Directory
- Issue requests to the appropriate endpoint using the Connector Server (or Superagent)

Several server plug-ins exist in the Server framework. The most common server plug-in is called SASP (for Superagent Server Plug-in). SASP knows how to send LDAP requests to the Provisioning Directory and to the Connector Server. SASP uses parser tables to determine how to process requests from the client interfaces.

This section contains the following topics:

[SASP Object Processing](#) (see page 89)

[SASP Characteristics](#) (see page 90)

[The C++ Connector Server](#) (see page 90)

[How to Build an Agent Plug-In](#) (see page 92)

[Directory Information Tree \(DIT\)](#) (see page 100)

SASP Object Processing

Adding

SASP adds a new object to the Provisioning Directory and to the endpoint using the Superagent. The object in the Provisioning Directory contains fewer attributes than the object on the endpoint system.

Retrieving

To retrieve objects, SASP accesses the Provisioning Directory or the endpoint, depending on what is requested. Parser tables indicate to SASP where each attribute is stored and helps it determine where to locate the information as follows:

- If all the requested attributes are stored in the Provisioning Directory, the request is routed to the Provisioning Directory.
- If all the requested attributes are stored on the endpoint, the request is routed to the endpoint through a C++ Connector Server or Java Connector Server.

Updating

SASP updates the endpoint through a Connector Server and also updates the object, including its timestamp, in the Provisioning Directory.

Deleting

SASP deletes the object on the endpoint through a Connector Server and also deletes the object from the Provisioning Directory. If the object you are deleting has one or more inclusion objects associated with it, the inclusion objects are also deleted.

SASP Characteristics

SLAPD is a multi-threaded application. The Provisioning Server and server plug-ins have been designed to support multiple clients and multiple requests from a single client concurrently. In some cases where concurrency is not possible, the Provisioning Server or server plug-ins sequence actions to produce the correct behavior. This multi-threaded behavior is also found in other components, such as the Connector Server and Provisioning Manager.

All text data received or sent through LDAP requests is in UTF-8 format. Every Unicode character translates to one to three UTF-8 bytes. The ASCII 0-127 characters are unchanged in UTF8. For more information about UTF-8 character encoding, see *Internet RFC #2279*.

Some attribute values are case-sensitive and others are case-insensitive. The parser table provides the CASE property that indicates whether the attribute is case-sensitive or case-insensitive. The CASE property can also indicate whether a value is upper case or lower case. Object class names and attribute types in LDAP are always case-insensitive. Be careful with container objects named through the EXTNAME class-level property. Verify that the connector and the parser table agree on the capitalization.

The C++ Connector Server

The C++ Connector Server has no persistent storage for configuration information. It relies on the Provisioning Server to determine which connectors have been installed and which endpoints have been registered. The Provisioning Server keeps this information in the Provisioning Directory.

When the C++ Connector Server starts, no connectors are loaded, and therefore, the in-memory DIT data structure is empty.

SASP uses the information from the Provisioning Directory to send LDAP ADD requests to the C++ Connector Server (as well as the Java Connector Server) when adding each endpoint object that it needs to access. The following process takes place:

- Adding an endpoint object causes the superagent framework (C++ Connector Server) to load the agent plug-in DLL (connector) for the endpoint type.
- Adding an endpoint object causes the agent plug-in to populate the in-memory DIT with any fixed containers, such as account containers.

Connector Server Functions

The Connector Server provides several functions that help you write LDAP commands without knowing all the details of LDAP.

Factory Function	Usage
DEadd()	Performs an LDAP ADD.
DEdelete()	Performs an LDAP DELETE.
DEmodify()	Performs an LDAP MODIFY.
DEmodrdn()	Performs an LDAP RENAME on the RDN or container as needed for rename and move operations.
DEsearch()	Performs an LDAP SEARCH.
DEbind()	Performs an LDAP bind.

More information:

[Superagent Framework APIs](#) (see page 289)

Running Agent DLLs Using the VC++ Debugger

There are two slapd services with the following names:

- eta_slapd (slapd.exe)
- eta_connector (superagent.exe).

The slapd.exe is dependent upon superagent.exe. Admin server slapd listens to ldap 20389 and ldap/tls 20390, while the Connector Server listens to ldap 20402 and ldap/tls 20403.

Note: The Connector Server ports have been relocated, each one with its own ldap/tls port. Both services can be started/stopped using net start/stop eta_slapd and net start/stop eta_connector.

1. Stop the eta_slapd service and the eta_connector service.
2. Open a command prompt and change to the PSHOME/bin directory.
3. Enter the following at the command prompt:
msdev superagent.exe
4. Right-click superagent project.
5. Select the Debug tab.
6. Specify a working folder or leave it empty as the current working folder in PSHOME/bin.
7. Specify "Program arguments" as follows:

```
-f ../data/eta_connector.conf -h "ldap://localhost:20402"  
"ldaps://localhost:20403" "ldap://<host>:20402" ldaps://<host>:20403"
```
8. The eta_slapd service also needs to be started from msdev. Otherwise, starting eta_slapd will bring up eta_connector service as well. When two Connector Servers are running, there are port conflicts.

How to Build an Agent Plug-In

The most common scenario for building an agent plug-in includes the following steps:

1. Initialize the agent plug-in.
2. Declare object classes.
3. Construct an agent DIT.

How to Initialize an Agent Plug-in

An agent plug-in is a DLL that is loaded at runtime by the Connector Server. The Connector Server loads the DLL when it receives an LDAP ADD request from SASP with the DLL name of the endpoint.

When the Connector Server starts, no agent plug-ins are loaded. SASP reads configuration information from the Provisioning Directory, determines which endpoint and directories have already been added to the Provisioning Directory, and sends LDAP ADD requests to the Connector Server that loads agent plug-ins and adds directory objects for each endpoint. After a directory object is added, the agent plug-in can manage the endpoint.

When the DLL is loaded, the Connector Server calls the function `init_agent()`. The agent plug-in (DLL) must define an `init_agent()` function.

When the DLL is unloaded, the Connector Server calls the function `shutdown_agent()`.

Note: Client interfaces cannot read or manipulate directory objects until they are added to the Provisioning Directory.

When the Connector Server receives an LDAP ADD for a directory object, it performs the following:

1. Calls the factory function for the object class.
2. Calls the `DEinit()` method on the resulting `DMODirectoryEntry` object.
3. Links the `DMODirectoryEntry` object to its in-memory DIT.

The `DEinit()` method allocates entry local storage and creates an instant DIT subtree with its child nodes, if specified. A non-hierarchical connector, such as the sample SDK connector, creates its fixed container hierarchy inside the `DEinit()` method of its endpoint class. This makes all of the objects under the endpoint immediately accessible to SASP .

`init_agent()` Function

`int __declspec(dllexport) init_agent(DERepository **ppRep);`

This function is called immediately after the Connector Server loads the endpoint DLL. It returns an array list of all object classes defined for the endpoint type and a corresponding factory function for each object class.

After calling the `init_agent()` function, the Connector Server calls the factory functions for the endpoint object and places the returned `DEDirectoryEntry` object into its in-memory DIT. The Connector Server also initializes the `DEDirectoryEntry` object by calling the `DEinit()` method.

`shutdown_agent()` Function

`int __declspec(dllexport) shutdown_agent();`

This function is called when an endpoint is unloaded.

The agent plug-in DLL can be unloaded because of a endpoint LDAP DELETE request or because the Connector Server shutdown. In these cases, the Connector Server calls the function `shutdown_agent()` to allow the agent plug-in a chance to clean up.

The agent plug-in (DLL) can define a `shutdown_agent()` function to perform agent plug-in resource clean up, etc. If the agent plug-in does not need to perform agent plug-in clean up on shutdown, then it does not need to define a `shutdown_agent()` function.

How to Declare an Object Class

Attributes of a directory object are defined by the subclasses derived from the `DMODirectoryEntry` base class. For example, the directory entry types for the SDK connector are:

- `CSDKNamespace`
- `CSDKDirectory`
- `CSDKGroupFolder`
- `CSDKAccountFolder`
- `CSDKGroup`
- `CSDKAccount`.

The C++ code for this example is similar to the following:

```
class SDKNamespace:           public DMODirectoryEntry
class SDKDirectory:           public DMODirectoryEntry
class SDKAccountFolder:       public DMODirectoryEntry
class SDKGroupFolder:         public DMODirectoryEntry
class SDKAccount:             public DMODirectoryEntry
class SDKGroup:               public DMODirectoryEntry
```

Each directory entry type can declare its own variables and functions that manipulate the member functions. The member functions invoke the superagent's LDAP operations on the directory object.

These member functions are declared as virtual. A virtual function is an optional function. If you do not implement one of the member functions, a default implementation of the function provided in the parent class is called instead.

For example, the SDK Account is declared as follows:

```

class CSDKAccount: public DMOFlatDirectoryEntry
{
public:
    CSDKAccount();
    CSDKAccount(
        UTF8          *pszObjectClass

    );

    CSDKAccount(
        UTF8          *pszObjectClass,
        UTF8          *pszRdnType,
        UTF8          *pszRdnValue

    );

    // LDAP Operations
    virtual int DEinit(
        DMO_LDAP_Entry *pEntry,
        DMOMessage      *pStatusMsg

    );

    virtual int DEadd(DMOAddOp* pAddOp);
    virtual int DEbind(DMOBindOp* pBindOp);
    virtual int Edelete(DMOMDeleteOp* pDeleteOp);
    virtual int Emodify(DMOMModifyOp* pModifyOp);
    virtual int Emodrdn(DMOModrdnOp* pModrdnOp);
    virtual int Esearch(DMOSearchOp* pSearchOp);
    . . .
};

```

How to Construct an Agent DIT

Use the following steps to construct an agent plug-in DIT:

1. Instantiate DMODirectoryEntry objects and establish parent-child relationships when you add a new directory object.
2. Use the DEinit() function to initialize objects in a directory that is being created in the DIT.

Note: All container nodes appear in the DIT, but leaf nodes appear only once per container for all the objects in the container.

The subclasses in the SDK Namespace are:

```
class CSDKNamespace:           public DMODirectoryEntry
class CSDKDirectory:           public DMODirectoryEntry
class CSDKGroupFolder:         public DMOFlatDirectoryEntry
class CSDKAccountFolder:       public DMOFlatDirectoryEntry
class CSDKGroup:               public DMOFlatDirectoryEntry
class CSDKAccount:             public DMOFlatDirectoryEntry
```

Each class provides a set of LDAP-like functions that implement LDAP operations for the corresponding object.

Example - DEinit() Function

The following DEinit() function shows how the initialization of the eTSDKDirectory object creates the objects in the DIT.

Note: All container nodes appear in the DIT, but leaf nodes appear only once per container and act as a placeholder for all the accounts.

```
int
CSDKDirectory::DEinit(
    DMO_LDAP_Entry      *pEntry,
    DMOMessage           *pStatusMsg
)
{
    DMODirectoryEntry    *pAccount      = NULL;
    DMODirectoryEntry    *pAccountFolder = NULL;
    DMODirectoryEntry    *pGroup        = NULL;
    DMODirectoryEntry    *pGroupFolder  = NULL;
    UTF8                 *pszComment    = NULL;
    UTF8                 *pszDir        = NULL;
    UTF8                 *pszHost       = NULL;
    UTF8                 *pszIp         = NULL;
    int                  rc              = LDAP_SUCCESS;

    /*
    || Check parameter.
    */
    if (pEntry == NULL) {
        rc = LDAP_OPERATIONS_ERROR;
        goto exit;
    }
}
```



```
/*
|| Each directory should register a logging component.
|| First, create the pLogComp member variable. Then
|| initialize it.
*/
registerLogComponent();          // create pLogComp
if (pLogComp) {
    pLogComp->registerLogComponent(pEntry, pStatusMsg);
}

/*
|| Get the directory name.
*/
pszuDir = pDN->getRDNvalue();
if (pszuDir == NULL) {
    rc = LDAP_INVALID_DN_SYNTAX;
    goto exit;
}

/*
|| Get required attributes.
*/
pszuIp = pEntry->getAttr("eTSDKIpAddress");
if (pszuIp == NULL) {
    rc = LDAP_NO_SUCH_ATTRIBUTE;
    pStatusMsg->SetMessage8(
        "Missing required attribute: eTSDKIpAddress");
    goto exit;
}
m_pszuIp = ch_strdup(pszuIp);

/*
|| Get optional attributes.
*/
pszuHost = pEntry->getAttr("eTSDKHost");
if (pszuHost) {
    m_pszuHost = ch_strdup(pszuHost);
}
pszuComment = pEntry->getAttr("eTSDKComments");
if (pszuComment) {
    m_pszuComment = ch_strdup(pszuComment);
}
```

```
/*
|| Each directory must build a Directory Information Tree, DIT.
|| The structure of the DIT is namespace specific. This sample
|| namespace has a simply DIT as follows:
||
||      Directory
||          AccountFolder
||              Account
||          GroupFolder
||              Group
||
|| If this method return a success, the superagent will add
|| the namespace DIT to a main DIT managed by the superagent.
*/

/*
|| Build the AccountFolder and Account node.
*/
pAccountFolder = new CSDKAccountFolder();
if ((pAccountFolder == NULL) ||
    (pAccountFolder->Deinit(NULL, pStatusMsg) != LDAP_SUCCESS))
{
    rc = LDAP_UNAVAILABLE_CRITICAL_EXTENSION;
    goto exit;
}
pAccount = new CSDKAccount("eTSDKAccount");
if ((pAccount == NULL) ||
    (pAccount->Deinit(NULL, pStatusMsg) != LDAP_SUCCESS))
{
    rc = LDAP_UNAVAILABLE_CRITICAL_EXTENSION;
    goto exit;
}
```

```
/*
|| Build the GroupFolder and Group node.
*/
pGroupFolder = new CSDKGroupFolder();
if ((pGroupFolder == NULL) ||
    (pGroupFolder->DEinit(NULL, pStatusMsg) != LDAP_SUCCESS))
{
    rc = LDAP_UNAVAILABLE_CRITICAL_EXTENSION;
    goto exit;
}
pGroup = new CSDKGroup("eTSDKGroup");
if ((pGroup == NULL) ||
    (pGroup->DEinit(NULL, pStatusMsg) != LDAP_SUCCESS))
{
    rc = LDAP_UNAVAILABLE_CRITICAL_EXTENSION;
    goto exit;
}
/*
|| Perform namespace specific initialization.
*/
m_pTarget = new Target(pszuDir);
rc = m_pTarget->Init(pStatusMsg);
if (rc != LDAP_SUCCESS) {
    rc = LDAP_OPERATIONS_ERROR;
    goto exit;
}

/*
|| Adding the node to the DIT should be the last step.
*/
/*
|| Add AccountFolder node as a child of the Directory node (this).
|| Then add the Account node as a child of the AccountFolder node.
*/
this->adopt_child(pAccountFolder);
pAccountFolder->adopt_child(pAccount);
```

```
/*
|| Add GroupFolder node as a child of the Directory node (this).
|| Then add the Group node as a child of the GroupFolder node.
*/
this->adopt_child(pGroupFolder);
pGroupFolder->adopt_child(pGroup);

exit:

if (rc != LDAP_SUCCESS) {
    if (pAccount) {
        delete pAccount;
    }
    if (pAccountFolder) {
        delete pAccountFolder;
    }
    if (pGroup) {
        delete pGroup;
    }
    if (pGroupFolder) {
        delete pGroupFolder;
    }
}
return rc;
}
```

Directory Information Tree (DIT)

The parser table syntax lets you define many possible DIT organizations. Because of this, each endpoint must follow guidelines when integrating its DIT into the Provisioning Server. These guidelines apply to the following:

- Naming endpoints, object classes, objects, and attributes
- Creating the required objects in the Provisioning Directory
- Defining objects

Important: Verify that the schema and DIT for the agent plug-in are structured the same way as the GUI plug-in parser table. If these are not structured the same way, communication problems will occur between the Provisioning Server and the endpoint.

Naming Endpoints, Object Classes, Objects, and Attributes

Guidelines are needed to prevent naming conflicts when creating names for endpoints, class objects, classes, and attributes.

Endpoints

Use a unique four-letter abbreviation for the name of each endpoint. You should follow the same naming convention for a DIT as you do for your parser table.

More information:

[Building Parser Tables](#) (see page 105)

Object Classes

Use a unique naming attribute for all object classes. This unique attribute tells the Server framework which object classes to manipulate and which server plug-ins to call.

Typically, you should add the string "Name" to the object class name when creating a naming attribute. For example, the naming attribute for the eTXXXXAccount object class is eTXXXXAccountName.

Objects and Attributes

Begin the name of all objects and attributes with the characters eTXXXX, where XXXX is the four-letter abbreviation for the namespace (connector).

Creating Required Objects

The Provisioning Server requires you to create two objects in the Provisioning Directory for each connector that you add. These objects are the eTXXXXNamespace and eTXXXXPolicyContainer objects.

Note: You can create one or more eTXXXXPolicy objects in the eTXXXXPolicyContainer object to provide your users with predefined policies. An example of this is provided in the SDKPop.C file. See the sample connector file SDKPop.C for a sample of a proper initialization program.

eTXXXXNamespace

The eTXXXXNamespace object should be placed directly in the server's domain component. It should be defined with the following attributes:

objectclass

Set to eTXXXXNamespace.

eTNamespaceName

Set to the value that you provide with the NAMESPACE statement in your parser table.

eTNamespaceType set to eTXXXXDirectory

Object class of your directory object.

eTAgentNamespaceClass

Set to the eTXXXXNamespace, which is the object class that your agent plug-in defines as its namespace (connector) class.

Note: The superagent framework requires each namespace (connector) to be of a unique class, whereas the Server framework expects all namespaces to use the common object eTNamespace class.

eTAgentPluginDLL

Set to XXXXNamespace.DLL, which is the DLL that contains your agent plug-in.

eTXXXXPolicyContainer

The eTXXXXPolicyContainer object should be placed directly in the eTNamespaceName=CommonObjects object.

It should be defined with the following attributes:

- objectclass set to eTXXXXPolicyContainer.
- eTXXXXPolicyContainerName set to the EXTNAME value you specified for the eTXXXXPolicyContainer class in your parser table.

Defining Objects

How to Define a Server Plug-in with a NAMESPACE Statement

To define SASP as the server plug-in for your namespace (connector), specify the following in the NAMESPACE statement:

SMPLUGINDLL=SASPRegister

How to Define an Object Class with a CLASS Statement

Do the following when defining object classes using the CLASS statement:

- If the directory you want to manage is a non-hierarchical directory, define container objects for each object you are managing.

For example, if you are managing accounts with the eTXXXXAccount object class, define the eTXXXXAccountContainer object class and set the DITPARENTCLASS property of the eTXXXXAccount and eTXXXXAccountContainer classes to eTXXXXAccountContainer and eTXXXXDirectory, respectively, to indicate that the accounts must be placed in this container.

- For every class you define, use the #include statement to include the common objects file called include/pti/comboject.pti. This file picks up definitions for attributes that apply to every class in the Provisioning Directory.

Note: If you are defining an object, such as a resource that is not stored in the Provisioning Directory, you can omit the #include statement.

- When defining a connector-specific version of a base class, define the SUPERCLASS property. This property identifies the class as a modification to the base class. You should use a #include statement with the corresponding files to pick up attribute definitions from the base class.

- /include/pti/account.pti
- /include/pti/container.pti
- /include/pti/directory.pti
- /include/pti/policy.pti

- Namespace (connector) objects are defined using the common object class eTNamespace in the Provisioning Server. The superagent framework requires each agent plug-in to define the object with a connector-specific object class (eTXXXXNamespace).

To account for this difference, define the superagent namespace class in the parser table and provide this class name to the eTAgentNamespaceClass property value when you create the namespace node.

How to Set the DATALOCATION Property for the Attributes

For every attribute of every class defined in the parser table, you must provide the correct setting of the DATALOCATION property. The possible values for this property are db, agent, both, and none.

- The db and both values indicate that the attribute should be stored with the object in the Provisioning Directory.

- The agent and both values indicate the attribute should be stored with the object in the connector server.
- If no value is specified for an attribute, it is presumed to be db.

The DATALOCATION property has the following special uses:

- The DATALOCATION property on the naming attribute defines where the object is stored. If the naming attribute says the object is stored only on the connector server (that is, the object class naming attribute has a DATALOCATION setting of agent), then no attributes are written to the Provisioning Directory. If the naming attribute's DATALOCATION is anything other than BOTH, objects of this class will not be explored.
- If the DATALOCATION property on a naming attribute has a setting of db, then all attributes not marked with none will be stored with the object in the Provisioning Directory. This behavior is defined so that eTXXXXPolicy objects can share the same account attribute definitions as the eTXXXXAccount object, even though most of the account attributes will be defined with a DATALOCATION property of agent.
- DATALOCATION settings must become more restrictive as you descend in the DIT hierarchy. To store an object in the Provisioning Directory or on the connector server, the container in which the object resides must also be stored there. In other words, the DATALOCATION property is relevant on containers, such as eTXXXXDirectory and eTXXXXAccountContainer, as well as on leaf nodes, such as eTXXXXAccount. In general, all naming attributes for connector object classes (except policy, policy container, and resource classes) should have the DATALOCATION value both. Policy and policy container classes should use db, and resource classes should use agent.
- Except in the eTXXXXDirectory class, the DATALOCATION settings for non-naming attributes in all connector-specific classes should be set to agent. Directory objects are a special case because they do not correspond with an actual persistent object stored in the connector directory.

The directory attributes that are necessary for agent initialization should be stored in the Provisioning Directory also. For all other connector objects, store only the name of the object in the Provisioning Directory. Any other type of information, if stored in the Provisioning Directory, will become out of sync if changed outside the control of this Provisioning Server. Those changes may not be detected until the next exploration on the object.

The policycontainerclass property applies only to eTXXXXDirectory classes. You must set this property to the name of your policy container class (eTXXXXPolicyContainer) for default directory behavior to function. The default policy is stored in directory objects as a complete DN.

Chapter 8: Building Parser Tables

Parser tables are used to define schemas in the Provisioning Server. You can define the following:

- Name of your GUI and server plug-in DLLs
- Information for the client interface and the GUI plug-in that presents data in Provisioning Manager
- Information that permits the Provisioning Server to validate user input
- Organization of data in your endpoint type

This section contains the following topics:

[Parser Table Building Process](#) (see page 105)

[Naming Parser Table](#) (see page 107)

[NAMESPACE Statement](#) (see page 108)

[CLASS Statement](#) (see page 109)

[KEYWORD Statement](#) (see page 115)

[Search and Long-valued Attribute Issues](#) (see page 125)

Parser Table Building Process

The source for parser tables consists of keywords, properties, comments, and #include statements.

To create a parser table, follow these steps:

1. Define the endpoint type.
2. Define the classes in the endpoint type.
3. Define the attributes that are related to the objects in the class.

For example:

```
#  
# Namespace for SDK objects  
#  
NAMESPACE SDK Namespace  
    smplugindll=SASPRRegister
```

```
#####  
# SDK Directory Class  
  
CLASS SDKDirectory,eTSDKDirectory  
    predefined=no  
    containerclass=eTNamespace  
    win32dll=CASDKGUI  
    win32icon=IDI_SDK_DIRECTORY  
    Win32DefaultMenu=no  
    win32menu=IDR_SDKDIR_MENU  
    draggable=yes  
    expandable=yes  
    sheet=yes  
    superclass=eTDirectory  
    nameproperty=eTSDKDirectoryName  
    ditparentclass=eTNamespace  
    policycontainerclass=eTSDKPolicyContainer  
  
    child_class=eTGlobalUser  
    child_class=eTGlobalGroup  
    parent_class=eTSDKPolicy  
  
KEYWORD name  
    minabbrev=4  
    attribute=eTSDKDirectoryName  
    datalocation=both  
    editttype=string  
    editflag=yes  
    minlen=1  
    maxlen=100  
    minvalue=  
    maxvalue=  
    description=eTrust Admin Directory SDK Name  
    default=  
    asciionly=yes  
    verbreq=add  
    verbreq=!toupdate
```

```
#
# Get the common object attributes
#
#include "../ .././include/pti/comobject.pti"
#
# Get the common directory attributes (and the optional UseAdminCreds attribute)
#
#include "../ .././include/pti/directory.pti"
#include "../ .././include/pti/Directory_UseAdminCreds.pti"
```

Naming Parser Table

The standard naming convention for parser tables is XXXXParse.pty, where XXXX is the prefix assigned to each endpoint type. The source file must have extension pty, and #include files must have extension pti. After you compile your code, a file with extension ptt is created. The ptt file is the compiled binary format of the parser table. Manager and the Provisioning Server read this file at startup. See the SDKParse.pty sample file, which is provided with the SDK.

View the Contents of Binary PTT Files

Use the dumpptt.exe utility to view the contents of all binary PTT files that are currently installed in the PSHOME/DATA directory.

To view the contents of binary PTT files

1. Open a command prompt and change to the following directory:

```
PSHOME/DATA
```

2. Run the dumpptt.exe utility as follows:

```
dumpptt -f > dumpptt.out
```

To display a list of all the dumpptt command line options, enter the following:

```
dumpptt -h
```

NAMESPACE Statement

The first statement in your parser table must be a NAMESPACE statement. This statement encloses the entire body of the parser table.

Note: Although you can define multiple endpoint types in a parser table, it is much easier to define one parser table for each endpoint type.

The NAMESPACE statement has the following format:

```
NAMESPACE namespace_name  
SMPLUGINDLL=dll_name  
HOSTED=boolean_value
```

NAMESPACE namespace_name

Required keyword. The name you use to define the endpoint type. This name must match the name of the namespace object that you create. The *namespace_name* value is used when building the DN of your directory entries.

SMPLUGINDLL=dll_name

Required keyword. The name of the server plug-in that handles all classes defined in the connector.

Note: The only value you should use for *dll_name* is SASPRegister.

HOSTED=boolean_value

Optional keyword. The possible values are Yes or No. Default is No. If Yes is specified, the Provisioning Server creates accounts for this connector during user-to-role synchronization in a second pass. This permits any dependent operating system accounts to be created first.

CLASS Statement

The CLASS statement defines an object class in a endpoint type. The format of the CLASS statement is as follows:

```
CLASS user_friendly_name, ldap_objectclass_name
  [DITPARENTCLASS=ldap_objectname_of_parentclass]
  [SUPERCLASS=internal_name_of_superclass]
  [HIDDEN=boolean]
  NAMEPROPERTY=ldap_property_name[,name,name . . .]
  [WIN32DLL=dll_name]
  [WIN32ICON=icon_name]
  [WIN32SELECTEDICON=alt_icon_name]
  [WIN32MENU=menu_name]
  [WIN32TOOLMENU=tool_menu_name]
  [WIN32TOOLBAR=tool_bar_name]
  [WIN32HELP=help_file]
  [CONTAINERCLASS=internal_class_name]
  [DRAGABLE=boolean]
  [EXPANDABLE=boolean]
  [SHEET=boolean]
  [CHILD_CLASS=internal_class_name]
  [PARENT_CLASS=internal_class_name]
  [PREDEFINED=boolean]
  [EXTNAME=predefined_name]
  [POLICYCONTAINERCLASS=internal_class_name]
  [DELETABLE=boolean]
  [DELETERECURSIVE=boolean]
  [INITIALIZEAGENT=boolean]
  [ISBASECLASS=boolean]
  [AUTHOPS=privileges_list]
  [SEARCHABLE=boolean]
  [CREATABLE=boolean]
  [INDOMAIN=domain_spec]
  [SORTABLE=boolean]
  [MOVABLE=boolean]
  [RENAMABLE=boolean]
  [DynamicContainer=boolean]
```

internal_classname

Name of the class, which can be used in Batch Utility.

Required keyword.

ldap_objectclass_name

LDAP name of the class, which can be used in Batch Utility.

Required keyword.

Note: The *ldap_objectclass_name* can contain only alphanumeric characters and the dash (-) character. All other characters are invalid.

Typically, *internal_classname* is the same as *ldap_objectclass_name*, without the leading "eT". Classes also have a *user_friendly_name* (for more information, see EXTNAME).

DITPARENTCLASS=*ldap_objectname_of_parentclass*

Location of the class in the directory hierarchy. Optional keyword.

Multiple DITPARENTCLASS keywords can be specified for an object class.

If you do not specify a location, the class does not exist in the directory. For example, base classes such as eTAccount do not exist in the directory. However, they exist so they can be referenced as the SUPERCLASS of another class.

SUPERCLASS=*internal_name_of_superclass*

Name of the logical super class. Optional keyword except when defining directory, account, policy, or container objects.

No inheritance is implied. This keyword is used by Manager to relate the connector objects to their common object class. The following are the required *internal_name_of_superclass* values for objects of the specified type:

Object Type	Required Value
Directory	eTDirectory
Account	eTAccount
Policy	eTPolicy
Container	eTContainer

HIDDEN=*boolean*

Hides the class from Batch Utility, but does not hide it from an LDAP client. Also controls which containers and object classes are visible in the Directory Content dialog of the client interface.

Optional keyword. The possible values are yes and no.

NAMEPROPERTY=ldap_property_name [, name, name, . . .]

Naming attribute for the class and RDN. Required keyword.

Provisioning Server requires that each class have a unique naming attribute. The convention is to name each class by adding the word Name to the end of the class name.

WIN32DLL=dll_name

Name of the DLL that handles window processing for the class. Optional keyword.

WIN32ICON=icon_name

Name of the icon resource. Optional keyword.

WIN32SELECTEDICON=alt_icon_name

Name of the icon resource that appears when an object is selected. If not specified, the WIN32ICON appears. Optional keyword.

WIN32MENU=menu_name

Name of the menu resource. Optional keyword.

WIN32TOOLMENU=tool_menu_name

Name of the tool menu resource. Optional keyword.

WIN32TOOLBAR=tool_bar_name

Name of the toolbar resource. Optional keyword.

WIN32HELP=help_file

Name of the Windows help file. Optional keyword.

CONTAINERCLASS=internal_class_name

Name of a GUI container class where objects of this class reside. Multiple CONTAINERCLASS keywords can be specified for an object class. Optional keyword.

DRAGABLE=boolean

Indicates whether you can drag and drop objects of this class. Optional keyword. The possible values are yes and no. Default is No. The permitted drag targets are controlled by the PARENT_CLASS and CHILD_CLASS properties.

EXPANDABLE=boolean

Indicates whether you can expand objects of this class. Optional keyword. The possible values are yes and no. Default is No. Determines whether a box with a plus sign appears in the tree to permit node expansion.

SHEET=*boolean*

Indicates whether clicking an object of this class displays a property sheet. Optional keyword. The possible values are Yes and No. Default is No.

CHILD_CLASS=*internal_class_name*

Specifies a class that can have inclusions. The class specified is the child in the relationship. Can be specified multiple times for different classes. Optional keyword. There is no ownership or hierarchy intended with a parent and child relationship. Inclusion relationships mean two objects are related.

Note: A parent-child relationship can be defined in the parent, the child, or both classes of the parser table. If you are defining a parent-child relationship with a common object, you must define the inclusion in your class only.

PARENT_CLASS=*internal_class_name*

Specifies a class that can have inclusions defined to it. The class specified is the parent in the relationship. Optional keyword. Can be specified multiple times for different classes. There is no ownership or hierarchy intended with a parent and child relationship. Inclusion relationships mean two objects are related.

Note: A parent-child relationship can be defined in the parent, the child, or both class of the parser table. If you are defining a parent-child relationship with a common object, you must define the inclusion in your class only.

PREDEFINED=*boolean*

Indicates whether the class is used for a single object in a fixed place in the directory hierarchy. Optional keyword. The possible values are Yes and No. Default is No. Typically, this is used for container nodes. The value of the naming attribute for a predefined class is provided with the EXTNAME keyword.

EXTNAME=*predefined_name*

For predefined classes, this property defines the value of the naming attribute for the object of this class. For non-predefined classes, this property can be used to give your class a user-friendly external name that can appear in user messages. Optional keyword.

POLICYCONTAINERCLASS=*internal_class_name*

Class name of the predefined policy container for the connector. This property is applicable only to classes where SUPERCLASS is eTDirectory. Optional keyword.

DELETABLE=boolean

This property displays the Delete menu commands for objects of this class. Optional keyword. The possible values are Yes and No. Default is Yes. If No is specified, the Delete menu is suppressed for the objects of the class.

DELETERECURSIVE=boolean

If Yes is specified, a delete operation on an object of this container class will be permitted even if the container is not empty. Optional keyword. The possible values are Yes and No. Default is No. All subordinate entries in the Provisioning Directory will be deleted when the specified container is deleted.

INITIALIZEAGENT=boolean

If Yes is specified, SASP will send ADD operations to the agent plug-in after the superagent re-initializes objects of this class from the Provisioning Directory. Optional keyword. The possible values are Yes and No. Default is No. By default, only the eTXXXXNamespace and eTXXXXDirectory objects are initialized in this manner. If INITIALIZEAGENT property is Yes, the behavior described by DYNAMICCONTAINER is also presumed.

ISBASECLASS=boolean

Optional keyword. The possible values are yes and no. Default is No. The value is set to Yes for the pre-defined base classes eTAccount, eTContainer, eTDirectory and eTPolicy. The SCHEMAGEN utility generates schema items only for base classes and for objects accessible through the DITPARENTCLASS chain.

AUTHOPS=privileges_list

Comma-delimited list of the administrator privileges for this class. Optional keyword. The default value is "add,delete,modify,owner,read". The following privileges are available:

- add-allows LDAP ADD operations.
- delete-allows LDAP DELETE operations.
- modify-allows LDAP MODIFY operations.
- owner-allows assignment of privileges to global users or admin profiles to control access to objects of this class.
- read-allows receipt of attributes from an LDAP SEARCH operation. For a container object, it is also required to list its contents or to specify any attribute other than the object class and the naming attribute in search filters or in LDAP COMPARE operations.

SEARCHABLE=boolean

If Yes is specified, the Directory Content dialog lets administrators issue multi-level searches for objects of this class. If No is specified, searches for objects of this class must be done from the immediate parent container. Optional keyword. Default is Yes.

CREATABLE=boolean

If Yes is specified, the Directory Content dialog lets administrators add new objects of this class. If No is specified, the "new" function is disabled for objects of this class. Optional keyword. Default is Yes.

INDOMAIN=domain_spec

Controls whether objects of this class can appear in the Provisioning Server domain, in normal user domains, or in both. You will always use the default value for all of your object classes. Optional keyword. Default is user. The possible values are: none, eta, user, or both.

SORTABLE=boolean

If Yes is specified, Manager lets objects of this class be reordered in lists by sorting on one of its attributes. If No is specified, Manager retains the order that these objects are returned from the Provisioning Server. Optional keyword. Default is Yes. This property is useful for objects that are not stored in the Provisioning Directory (objects whose naming attribute includes datalocation=agent) if the order of the objects is significant. For objects that are stored in the Provisioning Directory, the agent-defined ordering is not retained.

MOVABLE=boolean

If Yes is specified, MOVE operations on objects of this class are permitted. Optional keyword. The possible values are Yes and No. Default is No. The MOVE operation moves an object from one container to another, so you should enable this property only for objects that can appear in different containers and only if your agent plug-in implements the move operation in its DEmodrnn() function for this class.

RENAMABLE=boolean

Default is No. If Yes is specified, RENAME operations on objects of this class are permitted. The RENAME operation changes the name of an object while leaving it in the same container. Optional keyword. The possible values are Yes and No. You should enable this property only if your agent plug-in implements the rename operation in its DEmodrnn() function for this class.

DYNAMICCONTAINER=boolean

Optional keyword. The possible values are yes and no. Default is No. If Yes is specified, one-level exploration of this container's parent container will not add this container to the Provisioning Directory. In hierarchical connectors, dynamic containers are placed in the Provisioning Directory only when they are explored, as opposed to when their parent container is explored.

If in Provisioning Manager, an administrator explores a container, the Explore dialog sends two requests to the Provisioning Server: an LDAP ADD of the container setting the eTEExploreUpdateEtrust attribute to "1" and an LDAP Search of the container specifying eTEExploreUpdateEtrust in the list of attributes to return. ADD is a special add that updates the Provisioning Directory without actually creating a new container on the managed directory, and SEARCH is the special search that explores the contents of the container. Dynamic sub-containers should not be added to the connector directory because by not checking them in the Explore dialog, the administrator indicates that the sub-container is to be ignored.

KEYWORD Statement

The KEYWORD statement defines an attribute of a class. The format of the KEYWORD statement is as follows:

```
KEYWORD user_friendly_name
[MINABBREV=integer]
ATTRIBUTE=internal_LDAP_attribute_name
[DESCRIPTION=report_value]
EDITTYPE=data_type
[EDITFLAG=boolean]
[MULTIVALUE=boolean]
[CASE=sensitivity]
[MINLEN=integer]
[MAXLEN=integer]
[MINVALUE=integer]
[MAXVALUE=integer]
[HIDDEN=boolean]
[DEFAULT=value]
[VALUE=value]
[EXCLUDE=value]
[CHARNOTALLOWED=characters]
[NOTALLOWSPACE=boolean]
[ASCIIONLY=boolean]
[DATALOCATION=stored_in]
[INCLUSIONCHILD=internal_class_name]
[INCLUSIONPARENT=internal_class_name]
[VERBREQ=limitation]
```

```
[GLOBALUSERATTRIBUTE=internal_LDAP_attribute_name]  
[ALLOWPROPAGATION=boolean]  
[POLICYSYNC=boolean]  
[AUTHOPS=privileges_list]  
[ACCOUNTATTRIBUTE=internal_LDAP_attribute_name]  
[ENCRYPTED=boolean]  
[ENCRYPTWITH=Internal_LDAP_attribute_name_list]  
[INCLUSIONTARGETCHECK=boolean]  
[ISBASEATTRIBUTE=boolean]  
[SEARCHABLE=boolean]  
[DEPENDSON=internal_LDAP_attribute_name]  
[OVERRIDE=boolean]
```

KEYWORD *user_friendly_name*

User-friendly name for the attribute. Required keyword. You can use this name in Batch Utility rather than the LDAP name. This name can change during localization.

MINABBREV=*integer*

Minimum number of characters that can be used to abbreviate the user-friendly attribute name in Batch Utility. Optional keyword.

ATTRIBUTE=*internal_LDAP_attribute_name*

Attribute name used internally by the Provisioning Server and LDAP. This name does not change during localization. Required keyword.

Note: The *internal_LDAP_attribute_name* can contain only alphanumeric characters and the dash (-) character. All other characters are invalid.

DESCRIPTION=*report_value*

Optional keyword. The Provisioning Server uses this value when reporting errors while validating attribute values.

EDITTYPE=data_type

Type of data for this LDAP attribute. Required keyword. A valid *data_type* is one of the following:

- STRING-A character string
- INT-An integer
- BOOL-A boolean value
- CHAR-A single character
- DATE-A date that is stored as an integer (the year minus 1900)
- TIME-A time that is stored as an integer (100ths of seconds since midnight)
- DN-A character string containing a valid LDAP DN
- ORWORD-Defines unusual or different bit settings. For example, Windows users and groups have different user rights associated with them. Each user right is assigned a different bit setting. Bit setting 00000001 is associated with the user right "Access this computer from network" and setting 00000002 is associated with "Act as part of the operating system."

Note: Data defined as orwords are stored efficiently but are difficult to search for using LDAP. Define any bits of the orword to be searched as a separate attribute type of bool.

- BINARY-A binary value
- INCREMENTAL-Makes any modify request for this attribute appear to be an incremental change instead of a replacement. The default is No.
- SyncRemoveValues-Allows or denies the permission to remove values in weak synchronization operations.
- OBSCURED-Determines whether attributes such as passwords are always displayed as asterisks as they are typed and when displaying properties. If this value is No, passwords and other attributes are not displayed as asterisks.
- DEPRECATED-Specifies that the attribute is no longer used. The default is No.

EDITFLAG=boolean_value

Indicates whether less than or greater than comparisons can be used with the attribute when it is used for search criteria. Optional keyword. The possible values are yes and no. Default is Yes.

MULTIVALUE=boolean_value

Indicates whether the attribute can be assigned more than one value. Optional keyword. The possible values are yes and no. Default is No.

CASE=*sensitivity*

Specifies whether case-sensitive or case-insensitive matching is to be done. Also permits the case to be automatically folded to uppercase or lowercase when adding or updating the attribute. Optional keyword. Default is insensitive.

The values can be insensitive, insensitive-upper, insensitive-lower, sensitive, sensitive-upper, and sensitive-lower.

MINLEN=*integer*

Minimum length of a string or binary value. Optional keyword. Default is 0.

MAXLEN=*integer*

Maximum length of a string in characters or binary value in bytes. Optional keyword. While this property is optional, you should specify it so that you can query Manager and set the maximum length for an edit box. This prevents administrators from entering too many characters in a field.

Note: Very long strings may not be searched properly as the field length for searching a regular field is limited to the first 106 or so bytes.

More information:

[Search and Long-valued Attribute Issues](#) (see page 125)

MINVALUE=*integer*

Minimum value for an integer attribute. Optional keyword. Default is 0.

MAXVALUE=*integer*

Maximum value for an integer attribute. Optional keyword.

HIDDEN=*Boolean*

Hides the attribute from Batch Utility, but does not hide it from an LDAP client. Optional keyword. The possible values are yes and no. Default is No.

DEFAULT=*value*

Provides a default value for an attribute if none is specified during an ADD. Optional keyword.

VALUE=*value*

Value for the attribute. Optional keyword. Multiple properties can be specified for a single attribute. This property has different formats dependent on the EDITTYPE. It defines valid property values, alias values, or bit settings when defining or words. When the alias value is specified, it is converted to the real value before storing. The following table describes the valid values for each EDITTYPE.

EDITTYPE	Valid Values
BOOL	A numeric value (0 or 1), optionally followed by a comma and display string, such as Yes, No, True, False, Allow, or Deny. Batch Utility and LDAP accept either the numeric value or the string value. Batch Utility and Log Viewer display the string value if specified. Manager can retrieve the string values as well.
INT DATE TIME	A numeric value. This can be a regular expression, which accepts some values but not others.
STRING	A regular expression or a replacement string followed by an alias string. Either value is accepted, but only the replacement string is passed to the back end.
CHAR	A regular expression or a replacement character followed by an alias string. Either value is accepted, but only the replacement character is passed to the back end. Batch Utility and Log Viewer display the alias string value. It is very common for GUI plug-ins to build a drop-down list of acceptable values using the alias string.
ORWORD	Three values separated by commas: a bit value, a minimum abbreviation, and a string value. The bit value must be a hex string, such as 0800. The minimum abbreviation specifies the minimum number of characters in the string value that must be specified. This can be omitted to indicate that a full string must be entered. The string value specifies the value that may be specified by Batch Utility for the setting. Manager can query this as well. Batch Utility accepts and displays the string values associated with the bit settings.
BIN	Any binary value.
INCREMENTAL	No (the default) or Yes If Yes, the Admin server upon receipt of a modify request for this attribute, substitutes for any replace-mode modification item of the attribute the equivalent add and delete modification items so the changes appear to have been received as incremental changes instead of as a replacement set of values.
SyncRemoveValues	No (the default) or Yes If Yes, weak synchronization operations can remove values. A simple multivalued, policysync attribute such as group memberships of an account would typically define set this property to Yes to enable weak synchronization to remove group membership prescribed by a policy when that policy is removed from the account. Note: This property is relevant only if the multivalue property and the policysync property are Yes.

EDITTYPE	Valid Values
OBSURED	No (the default) or Yes Set the value Yes for attributes such as passwords that should always be displayed as asterisks as they are typed and when displaying properties, obscuring the value and whether or not the attribute is set.
DEPRECATED	No (the default) or Yes Set the value to Yes for attributes that are no longer used. Deleting items from the schema is difficult, but marking attributes as deprecated ensures that these attributes are not processed by clients or server

EXCLUDE=value

Specifies a regular expression matching values that are not considered valid. This is useful when there are exception values that are not permitted. Optional keyword. Multiple EXCLUDE properties can be specified for a single attribute.

CHARNOTALLOWED=characters

Characters that are not permitted anywhere in the value. This value is case-insensitive. Optional keyword.

NOTALLOWSPACE=boolean

Specifies whether embedded spaces are not permitted in a value. A user ID attribute is a candidate for this property. Optional keyword. The possible values are yes and no. Default is No.

ASCIIONLY=boolean

Specifies that only ASCII 127 characters can be specified. Optional keyword. The possible values are yes and no. Default is No.

DATALOCATION=stored_in

Location where objects of this class are stored. Optional keyword. The default value is DB. This indicates whether the server plug-in should access the connector server, the Provisioning Directory, both, or neither. A valid *stored_in* value is one of the following:

- AGENT-Namespace (connector) server
- DB-The Provisioning Directory
- BOTH-Connector server and Provisioning Directory
- NONE-Neither the connector server nor the Provisioning Directory

INCLUSIONCHILD=internal_class_name

Specifies that the attribute contains the names of related objects in the specified class. This lets the ETACLDAP client library simulate that inclusion objects are being used while actually manipulating values of this multi-valued attribute. The specified object class is the child class in the inclusion relationship. Optional keyword.

INCLUSIONPARENT=internal_class_name

Specifies that the attribute contains the names of related objects in the specified class. This lets the ETACLDAP client library simulate that inclusion objects are being used while actually manipulating values of this multi-valued attribute. The specified object class is the parent class in the inclusion relationship. Optional keyword.

VERBREQ=*limitation*

Specifies which requests require the attribute and which do not permit the attribute. Optional keyword. Multiple VERBREQ properties can be specified for a single attribute.

The possible values are:

- ADD-Specifies that the attribute is required when an object of this class is created using an LDAP ADD operation.
- !ADD-Specifies that the attribute is not permitted during LDAP ADD requests.
- TOUPDATE-Specifies that the attribute is required when an object of this class is modified using an LDAP MODIFY request.
- !TOUPDATE-Specifies that the attribute is not permitted during LDAP MODIFY requests.

GLOBALUSERATTRIBUTE=internal_LDAP_attribute_name

Applies only to accounts with SUPERCLASS=eTAccount. This property identifies the relationship between an account and a global user. If, during the correlation process, a new global user must be created for the account, the global user attribute that is identified here is initialized to have the same attribute value as the account. If the value of this property is eTGlobalUserName or eTFullName, the correlation process compares this attribute against the global user attribute to decide if an existing global user matches an account of this object class. If no attribute has GLOBALUSERATTRIBUTE set to eTGlobalUserName, the NAMEPROPERTY of the account class is used instead. Optional keyword.

ALLOWPROPAGATION=boolean

Applies only to accounts with SUPERCLASS=eTAccount. This property identifies which account attributes to update as a result of a policy change. If No is specified, the attribute is not updated. This property is ignored on read-only attributes. Optional keyword. Default is Yes.

POLICYSYNC=boolean

Applies only to accounts with SUPERCLASS=eTAccount. This property identifies which account attributes should be updated during account synchronization. This property is ignored on read-only attributes and attributes where ALLOWPROPAGATION is No. Account synchronization updates account attributes to be in compliance with the account's policies. For each multi-valued POLICYSYNC attribute, the account is made in compliance with a policy by ensuring that all values from the policy are included in the values of the attribute. For each single-valued POLICYSYNC attribute, the account is made in compliance with a policy by ensuring that the value of the account attribute is greater than or equal to the value indicated by the policy. For ORWORD POLICYSYNC attributes, the policy settings are OR'ed into the account attribute value. Optional keyword. Default is Yes if MULTIVALUE is Yes and No if MULTIVALUE is No.

AUTHOPS=privileges_list

This property is a comma-delimited list that defines the privileges one can assign for controlling access to this specific attribute on objects of this class. The MODIFY privilege lets an administrator set values of this attribute during LDAP ADD or MODIFY operations. The READ privilege permits the attribute's value to be accessed or returned during the processing of LDAP COMPARE or SEARCH operations. Optional keyword. Default is no values.

ACCOUNTATTRIBUTE=internal_LDAP_attribute_name

Applies only to classes with superclass=eTPolicy. If not specified, it defaults to the value of the attribute property. It defines which account attribute corresponds to this policy attribute. Typically, account and policy attributes are named the same; but if they are not, use this property to make the mapping explicit. The Provisioning Server will then create accounts from policies or propagate policy changes to accounts using this mapping rule. Optional keyword.

ENCRYPTED=boolean

If Yes is specified, the attribute is encrypted before being stored in the Provisioning Directory, and the Provisioning Server does not return the value (encrypted or in the clear) to any LDAP client. This value is useful only for attributes whose DATALOCATION property is set to BOTH. It is usually used in classes whose superclass is eTDirectory as a way of storing a connect password for the directory. The Provisioning Server decrypts the value before agent initialization so that the agent plug-in has the clear password required to connect to the connector directory. Optional keyword. The possible values are yes and no. Default is No.

ENCRYPTWITH=internal_LDAP_attribute_name_list

The value is a comma-separated list of LDAP attribute names of other attributes for the same object. The ENCRYPTWITH property is only relevant when the ENCRYPTED property is Yes. The named attributes will be combined with the encrypted attribute before it is encrypted so as to tie the encrypted value to the current values of those other attributes. When the attribute is later decrypted, the decryption will fail if the then-current values of those attributes are not the same as the values that are included in the decrypted block of data.

The ENCRYPTWITH property is useful when storing a password used to authenticate to a remote service or endpoint system being managed by the Provisioning Server. By including identifying information such as hostname and port number as attributes to be encrypted with the value, you ensure that the password is only sent to the service originally designated. Anyone changing what service the password is to be sent to would need to re-supply the password so that the Admin server could encrypt that newly supplied password with the current values of the ENCRYPTWITH attributes. Optional keyword.

INCLUSIONTARGETCHECK=boolean

Identifies an inclusion relationship specified with INCLUSIONPARENT or INCLUSIONCHILD for which the agent plug-in requires a read of the object prior to setting an agent attribute of another class to refer to this object. This property is used by the eTaLdapAddInclusion() library function which GUI plug-ins can use to treat intra-directory object relationships as simulated inclusions. Optional keyword. The possible values are yes and no. Default is No.

SBASEATTRIBUTE=boolean

Identifies an attribute that is inherited from the base class. All attributes defined in the included parser table files in include/pti/*.pti (for example, account.pti, policy.pti, and so on) are marked with this property so that SCHEMAGEN uses these attributes as part of the base schema and not a part of the connector's subclass. Optional keyword. The possible values are yes and no. Default is No.

SEARCHABLE=boolean

If Yes is specified, Manager lets administrators issue advanced searches for objects of this object class using this attribute in the filter. Optional keyword. The possible values are yes and no. Default is Yes.

Note: Very long strings may not be searched properly as the field length for searching a regular field is limited to the first 106 or so bytes.

More information:

[Search and Long-valued Attribute Issues](#) (see page 125)

DEPENDSON=internal_LDAP_attribute_name

This property is relevant for account object classes only. If specified, the value indicates the internal LDAP attribute name of another account attribute. This property may be specified multiple times if one attribute depends on multiple other attributes. The DEPENDSON property is used when the account attribute is stored in the Provisioning Directory. For account objects, the Provisioning Server will store any account attribute whose DATALOCATION property is DB or BOTH in the Provisioning Directory, as well as any attribute that is configured as relevant to Correlate, CreateUsers or UpdateUsers phases of exploration. Storing account attributes needed by these phases of exploration lets those phases run more quickly, thus, obtaining needed information from the Provisioning Directory without obtaining the data from the connector directory again. Once an attribute is stored in the Provisioning Directory, any change to the attribute through the Provisioning Server will update both the Provisioning Directory and the actual account in the connector directory.

The DEPENDSON property tells the Provisioning Server to update this attribute in the Provisioning Directory, if stored there, whenever the attribute to which this attribute depends is modified. The server will apply the change to the depended-on account attribute in the connector directory first, then read this attribute from the connector directory in order to write this attribute to the Provisioning Directory. As a special case of DEPENDSON, one can indicate that an attribute depends upon itself. This is useful if the attribute value looks different upon read than it did upon write. In that case, the value to be written to the Provisioning Directory should be the value read from the connector directory not the value written to the connector directory. Optional keyword.

OVERRIDE=boolean

If Yes is specified, the definition that follows does not define a new attribute, but updates a prior attribute definition. When overriding an attribute definition, include the KEYWORD line to indicate which attribute is being overridden, include the line "OVERRIDE=Yes", and include additional properties that replace the settings for this property. Typically, the OVERRIDE property is used to customize the definitions provided in the base class' PTI file. Common properties to override include DATALOCATION, DESCRIPTION, VALUE, MAXLEN, and CHARSNOTALLOWED. You cannot override a property that affects the generated schema definitions, so you cannot change whether the attribute is case-sensitive or change the EDITTYPE or ATTRIBUTE properties. Using the OVERRIDE property is preferable to copying the definitions from the base PTI file into your own PTI or PTY file as CA extensions made to the base PTI file later do not need to be copied to your PTI or PTY file. Optional keyword. The possible values are yes and no. Default is No.

Search and Long-valued Attribute Issues

Log warnings may appear when you try to search attributes with long values. Problems occur when searching for values whose first differences appear only after the first 106 or so bytes. These warnings may be eliminated by indicating the attribute as non-searchable or creating a wide index for it.

Indicating an attribute as non-searchable reduces functionality as it may make it impossible to test values for certain attributes.

Creating a wide index may slow performance, and the use of wide indexes should be minimized, and only considered when an attribute can frequently have long values that are identical over their approximately first 106 bytes.

More information:

[Declare an Attribute as Non-searchable](#) (see page 125)

[Create a Wide Index](#) (see page 126)

Declare an Attribute as Non-Searchable

Perform the following procedure to declare an attribute as non-searchable.

To declare an attribute as non-searchable

1. In the KEYWORD statement, set the SEARCHABLE tag to No.
2. Modify the eTDir configuration, as follows:
 1. Access the %DXHOME%/Config/Database/ folder, and copy the eTrustAdmin.dxc file to create a backup/restore file.
 2. Edit the eTrustAdmin.dxc file with a text editor, adding the name of the non-searchable attribute after the parameter "set not-searchable =", separating the relevant attribute names by a comma (,).
 3. Save the eTrustAdmin.dxc file.
 4. Restart the eTrustAdmin dxserver service.
 5. Access the PSHOME/bin/ folder, and copy the etaindex.bat file to create a backup/restore file.
 6. Edit the etaindex.bat file with a text editor, adding the name of the non-searchable attribute at the end of the line "dxindexdb etrustadmin +notSearchable...".

7. Save the etaindex.bat file.
8. Restart the eTrustAdmin dxserver service, and run etaindex.bat.

Note: Running etaindex.bat is not required when the attribute is new, and has no stored value.

Create a Wide Index for an Attribute

When an attribute with long values has been defined as searchable, you must create a wide index for it.

To create a wide index for an attribute

1. Access the %DXHOME%/Config/Database/ folder and duplicate the eTrustAdmin.dxc file to create a backup/restore file.
2. Edit the eTrustAdmin.dxc file with a text editor, adding to the parameter "set index-wide =" the name of the relevant attribute, separating it from other attribute names with a comma.
3. Save the eTrustAdmin.dxc file.
4. Restart the eTrustAdmin dxserver service.
5. Access the PSHOME/bin/ folder, and copy the etaindex.bat file to create a backup/restore file.
6. Edit the etaindex.bat file with a text editor, adding the name of the relevant attribute at the end of the line "xindexdb etrustadmin +wide...".
7. Save the etaindex.bat file.
8. Restart the eTrustAdmin dxserver service, and run etaindex.bat.

Note: Running etaindex.bat is not required when the attribute is new, and has no stored value.

Chapter 9: Building a GUI Plug-In

You can extend Provisioning Manager so that it can manage additional endpoints. If you want to extend it, you must build a GUI plug-in and define it to the GUI framework.

Although the GUI framework provides much of the required functionality in the Provisioning Server, your GUI plug-in must define property pages and other objects that appear in Provisioning Manager.

To build a GUI plug-in:

1. Build a parser table for the GUI plug-in.
2. Create a GUI callout module.
3. Define GUI objects.
4. Create the code for your property sheets and property pages.
5. Build and link the plug-in into the GUI framework.
 - Add objects to the Provisioning Directory.
 - Build help files.
6. Install and test the GUI plug-in.

Note: Each GUI plug-in must have a corresponding agent plug-in.

This section contains the following topics:

[Build a Parser Table for the GUI Plug-In](#) (see page 127)

[GUI Callout Module](#) (see page 128)

[GUI Objects Definition](#) (see page 139)

[Property Sheets and Property Pages](#) (see page 142)

[Building and Linking the GUI Plug-in](#) (see page 161)

[Install and Test the GUI Plug-in](#) (see page 162)

Build a Parser Table for the GUI Plug-In

When adding a new endpoint type, create a parser table that defines each of the objects in your endpoint type. Parser tables describe the content and tree hierarchy of the objects in your endpoint type.

Namespace Class

Every endpoint type should have a namespace class defined for it. To define a namespace class, define all the class-level properties and their attributes in the parser table for the endpoint type.

Directory Class

Every endpoint type should have a directory class defined for it. To define a directory class, define all the class-level properties and their attributes in the parser table for the directory.

Tree Hierarchy

Each class, except the namespace class, must be defined in the parser table using the CONTAINERCLASS property. By doing this, you define the hierarchy of the objects as they appear in Manager.

Samples for defining classes can be found in the SDKParse.pti, Account_Property.pti, directory.pti, and Group_Property.pti files.

The CASDKCLS.h file provides information on the different #defines for the class and attribute names in the sample. This file is used with the sample property pages and property sheets to ensure consistency for the attribute and class names between the parser table and the GUI code.

More information:

[Building Parser Tables](#) (see page 105)

GUI Callout Module

The GUI callout module is a source file that contains entry points to your GUI plug-in. By using these entry points, your GUI plug-in can manage certain events that can take place in Manager, such as dragging and dropping. The callout module has the following entry points:

GetPropertySheetID

Called when an administrator selects an object that has a property sheet associated with it. There is no default action. Your GUI plug-in must supply this function.

When the GUI framework calls this function, it passes the classname and the DN of the selected object. This function must then display the appropriate property sheet for the given classname and DN.

OnCommand

Called when an administrator selects a custom menu item. There is no default action for this callout. If you define custom menus, your GUI plug-in must supply this function.

IsValidDrag

Called when an administrator drags an object over another object in the tree. By default, the parser table is checked to make sure that the two classes have an inclusion relationship.

More information:

[Understanding Predefined Object Class](#) (see page 69)

DeleteEntry

Called when an administrator attempts to delete an object after confirming the deletion. The default action is to delete the entry.

CanDuplicateEntry

Determines whether a connector can duplicate an entry.

DuplicateEntry

Called when duplicating an object. The DLL should create and display a property sheet for the object using a manner similar to displaying property sheets. You can also clear attributes from the Global User property sheet when duplicating an object.

RenameEntry

Called when renaming an object.

CreateInclusion

Gives connectors the ability to implement non-default actions when inclusions must be created.

ChildrenToExplore

Used by hierarchical connectors that let containers be dynamically located. This function is used by the Explore/Correlate operation to help define the container tree.

Note: The file contains also obsolete entry points. These are not documented, and *must not be used*.

These entry points must be exported from your DLL which is typically done through a DEF file. The callout entry points are defined in the GUIEXIT.H file.

More information:

[GUI Callout Functions](#) (see page 279)

SDK Sample Connector Code

The following code defines the SDK sample namespace (connector).

```
extern "C"{
/*=====+

IsValidDrag

    An item was dragged on to an item in one of my classes. Check to see
    if we will accept the drag

Parameters:
    repository handle
    parser handle
    dragged class name
    dragged object name
    target class name
    target object name
    directory class for dragged object
    directory name for dragged object
    directory class for target object
    directory name for target object

Returns:
    TRUE/FALSE

Remarks:
```

```

+=====*/
int IsValidDrag (
    ETA_HANDLE hRepository.
    HANDLE      hptt.
    PWCHAR_T pszDragClass.
    PWCHAR_T pszDragObject.
    PWCHAR_T pszTargetClass.
    PWCHAR_T pszTargetObject.
    PWCHAR_T pszDragDirectoryClass.
    PWCHAR_T pszDragDirectoryName.
    PWCHAR_T pszTargetDirectoryClass.
    PWCHAR_T pszTargetDirectoryName
)
{
    return ETA_GUIEXIT_PASSTHRU;
}

/*=====+

```

GetPropertySheetID

A property sheet for an object is to be displayed

Parameters:

parent window (CWnd *)
 class name of object
 DN of object (base object for add)
 parent window handle (HWND)
 Client API handle
 Parser API handle
 Add ?
 display sheet stand-alone ?
 confirming deletes?
 use persistent selects?

Return Value:

a pointer to a valid property sheet

Remarks:

```

+=====*/
void * GetPropertySheetID(
    void*      pParentWnd.
    PWCHAR_T pszClassName.
    PWCHAR_T pszObjectDN.
    HWND      hGui.
    ETA_HANDLE hRepository.
)

```

```
HANDLE    hptt.  
BOOL_T    isNewObject.  
BOOL_T    isModal.  
BOOL_T    confirmDelete.  
BOOL_T    globalSelects.  
BOOL_T    isDupObject  
)  
{  
  if (wcscmp (pszClassName, SRD_ETASDKDIRECTORY) == 0){  
    return (void *)new CASDKDirectorySheet (  
      (CWnd *)pParentWnd.  
      pszClassName.  
      pszObjectDN.  
      hGui.  
      hRepository.  
      isNewObject.  
      isModal.  
      confirmDelete.  
      globalSelects.  
      isDupObject  
    );  
  }  
  else if (wcscmp (pszClassName, SRD_ETASDKACCOUNT) == 0){  
    return (void *)new CASDKAccountSheet (  
      (CWnd *)pParentWnd.  
      pszClassName.  
      pszObjectDN.  
      hGui.  
      hRepository.  
      isNewObject.  
      isModal.  
      confirmDelete.  
      globalSelects.  
      isDupObject  
    );  
  }  
  else if (wcscmp (pszClassName, SRD_ETASDKGROUP) == 0){  
    return (void *)new CASDKGroupSheet (  
      (CWnd *)pParentWnd.  
      pszClassName.  
      pszObjectDN.  
      hGui.  
      hRepository.  
      isNewObject.  
      isModal.  
      confirmDelete.  
      globalSelects.  
      isDupObject  
    );  
  }  
}
```

```
    }
    else if (wcscmp (pszClassName, SRD_ETASDKPOLICY) == 0){
        return (void *)new CASDKPolicySheet (
            (CWnd *)pParentWnd.
            pszClassName.
            pszObjectDN.
            hGui.
            hRepository.
            isNewObject.
            isModal.
            confirmDelete.
            globalSelects.
            isDupObject
        );
    }
    else if (wcscmp (pszClassName, SRD_ETASDKEXIT) == 0){
        return (void *)new CASDKExitSheet (
            (CWnd *)pParentWnd.
            pszClassName.
            pszObjectDN.
            hGui.
            hRepository.
            isNewObject.
            isModal.
            confirmDelete.
            globalSelects.
            isDupObject
        );
    }
    else {
        return (void *)new CAPropertySheet (
            (CWnd *)pParentWnd.
            pszClassName.
            pszObjectDN.
            hGui.
            hRepository.
            isNewObject.
            isModal.
            confirmDelete.
            globalSelects.
            isDupObject
        );
    }
}
```

```
/*=====+

GetPropertySheetID

A property sheet for an object is to be displayed

Return Value:

a pointer to a valid property sheet

Remarks:

=====*/
void * GetPropertySheetID(
    void*      pParentWnd.
    PWCHAR_T   pszClassName.
    PWCHAR_T   pszObjectDN.
    PWCHAR_T   pszObjectID.
    HWND       hGui.
    ETA_HANDLE hRepository.
    HANDLE      hptt.
    BOOL_T      isNewObject.
    BOOL_T      isModal.
    BOOL_T      confirmDelete.
    BOOL_T      globalSelects.
    BOOL_T      isDupObject
)
{
    if (wcscmp (pszClassName, SRD_ETASDKDIRECTORY) == 0){
        return (void *)new CASDKDirectorySheet (
            (CWnd *)pParentWnd.
            pszClassName.
            pszObjectDN.
#ifdef ID_SUPPORT_ON
            pszObjectID.
#endif
            hGui.
            hRepository.
            isNewObject.
            isModal.
            confirmDelete.
            globalSelects.
            isDupObject
        );
    }
    else if (wcscmp (pszClassName, SRD_ETASDKACCOUNT) == 0){
        return (void *)new CASDKAccountSheet (
            (CWnd *)pParentWnd.
```

```

        pszClassName.
        pszObjectDN.
#ifdef ID_SUPPORT_ON
        pszObjectID.
#endif
        hGui.
        hRepository.
        isNewObject.
        isModal.
        confirmDelete.
        globalSelects.
        isDupObject
    );
}
else if (wscmp (pszClassName, SRD_ETASDKGROUP) == 0){
    return (void *)new CASDKGroupSheet (
        (CWnd *)pParentWnd.
        pszClassName.
        pszObjectDN.
#ifdef ID_SUPPORT_ON
        pszObjectID.
#endif
        hGui.
        hRepository.
        isNewObject.
        isModal.
        confirmDelete.
        globalSelects.
        isDupObject
    );
}
else if (wscmp (pszClassName, SRD_ETASDKPOLICY) == 0){
    return (void *)new CASDKPolicySheet (
        (CWnd *)pParentWnd.
        pszClassName.
        pszObjectDN.
#ifdef ID_SUPPORT_ON
        pszObjectID.
#endif
        hGui.
        hRepository.
        isNewObject.
        isModal.
        confirmDelete.
        globalSelects.
        isDupObject
    );
}
else if (wscmp (pszClassName, SRD_ETASDKEXIT) == 0){

```

```
        return (void *)new CASDKExitSheet (
            (CWnd *)pParentWnd.
            pszClassName.
            pszObjectDN.
#ifdef ID_SUPPORT_ON
            pszObjectID.
#endif
            hGui.
            hRepository.
            isNewObject.
            isModal.
            confirmDelete.
            globalSelects.
            isDupObject
        );
    }
    else {
        return (void *)new CAPropertySheet (
            (CWnd *)pParentWnd.
            pszClassName.
            pszObjectDN.
#ifdef ID_SUPPORT_ON
            pszObjectID.
#endif
            hGui.
            hRepository.
            isNewObject.
            isModal.
            confirmDelete.
            globalSelects.
            isDupObject
        );
    }
}

/*=====+
OnCommand

    A command has been issued from the plugin menu

Parameters:

    Client API handle
    Parser API handle
    class name of selected object
    DN of selected object
    command ID

Return Value:
```


TRUE if message processed
 FALSE if message not processed

Remarks:

```
+=====*/
BOOL_T OnCommand (
    ETA_HANDLE hRepository.
    HANDLE      hptt.
    PWCHAR_T    pszClass.
    PWCHAR_T    pszName.
    UINT        uiCommand
)
{
    return TRUE;
}

/*=====+
```

RenameEntry

Rename an object

Parameters:

GUI framework handle
 Parser API handle
 Client API handle
 class name of selected object
 DN of selected object
 SDS buffer for GUI plug-in to send a message back to the framework

Return Value:

ETA_SUCCESS if successful
 ETA_GUIEXIT_PASSTHRU if GUI plug-in doesn't handle rename
 other ETA error code if error

Remarks:

```
+=====*/
int
RenameEntry(
    HWND      hGui.
    HANDLE     hptt.
    ETA_HANDLE hRepository.
    PWCHAR_T   pszClass.
    PWCHAR_T   pszDN.
```

```
void **    ppSds
)
{
    DWORD            dwError;
    int              iStatus;
    LPVOID           lpMsgBuf        = NULL;
    CString          sError;
    PWCHAR_T         pszMsg          = NULL;
    int              rc              = ETA_SUCCESS;

    /*
    || Need to switch context to the GUI plug-in.
    */
    HINSTANCE hOld = AfxGetResourceHandle();
    AfxSetResourceHandle(LoadLibrary(L"casdkgui"));

    /*
    || The RenameDlg dialog will handle the actual rename.
    || RenameDlg will send error message to the framework.
    || so it doesn't need to set the ppSds output argument.
    */
    RenameDlg dlg(hRepository, pszClass, pszDN);

    iStatus = dlg.DoModal();
    if (iStatus == -1) {
        rc = ETA_OPERATION_FAIL;

        /*
        || Can't display the dialog.
        */
        // ...Set error message.
        sError = L"Can't display SDK Rename dialog";

        // ...Get detail error number.
        dwError = GetLastError();

        // ...Get detail error message.
        if (FormatMessage(
            FORMAT_MESSAGE_ALLOCATE_BUFFER |
            FORMAT_MESSAGE_FROM_SYSTEM |
            FORMAT_MESSAGE_IGNORE_INSERTS,
            NULL,
            dwError,
            MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
            (LPTSTR) &lpMsgBuf,
            0,
            NULL))
        {
            // ...Add detail message.
```

```
sError += ": ";
sError += (LPTSTR) lpMsgBuf;

// Free the buffer.
LocalFree(lpMsgBuf);
}

/*
|| Pass back a message to the GUI framework, if the output
|| argument is provided.
*/
if (ppSds != NULL) {
    pszWMsg = (PWCHAR_T)((LPCTSTR)sError);

    /*
    || PutSdsMsg will allocate memory for the SDS buffer, if needed.
    || The GUI framework is responsible for freeing the memory.
    */
    PutSdsMsg(pszWMsg, ppSds);
}

/*
|| Pop up the message box.
*/
AfxMessageBox(sError);
}

/*
|| Restore GUI framework context.
*/
AfxSetResourceHandle(hOld);

return rc;
}

} /* extern "C" */
```

GUI Objects Definition

Use the Visual Studio dialog editor when defining GUI objects that will represent your namespace (connector) in Provisioning Manager. You can define menus and property pages. Property sheets can be defined to hold property pages, but do not have a dialog that can be edited by the VS Dialog Editor.

Resource ID Ranges

There are recommended restrictions on the range of IDs used for resources. These restrictions exist due to rules defined for the Windows platform, and to avoid conflict with the IDs that are already being used by the framework.

Note: Conflict between connectors is not a concern, so one namespace (connector) can use the same values as another, and this will not create a technical problem.

Resources are classified as follows::

- Command resources are used for menu commands. Safe range is from 32768 (0x8000) to 36863 (0x8FFF), inclusive.
- Control resources are used for controls on a page. Safe range is from 10 (0xA) to 12287 (0x2FFF) inclusive, and from 28672 (0x7000) to 32767 (0x7FFF) inclusive.
- Other resources are used for everything else, including strings, dialogs, and bitmaps. Safe range is from 10 (0xA) to 12287 (0x2FFF).

Note: Control and other resources are safe down to an ID value 10. However, it is recommended not to use values below 101, which is the default value used by Visual Studio.

Custom Menus

Each object can have one or more custom menu commands. The menu item is displayed in the Custom submenu in the object context menu, available when you right-click the object in the list view or from the Object menu in the main menu bar.

The following list provides important information about custom menus:

- All objects of the same class must use the same menu.
- The WIN32MENU property in your parser table class definition must be defined for every object with a menu.

For example, if you want the menu IDR_USER_MENU to appear when a user clicks an account for an object in your connector, add the following keyword definition to the parser table class definition for the account:

```
win32menu=IDR_USER_MENU
```

Property Sheets and Pages

A *property sheet* displays the properties and inclusions for an object. Property sheets consist of multiple property pages marked by tabs that display when an object is selected. Property pages let users manipulate a set of properties. You should group these properties logically, so each property page reflects one group.

When defining property sheets and their property pages, you should create a C++ class for each property sheet. This class is sub-classed from the `CAPropertySheet` class defined in `CAPropertySheet`. A related class, `CAPropertyPage`, is available for pages.

Notes:

- Multiple constructors are defined for `CAPropertySheet`. You should create two constructors in your property sheet, one of them compiled conditionally based on the setting of `ID_SUPPORT_ON`. As a general rule, only use these two constructors in a `CAPropertySheet` that uses the `isDupObject` parameter.
- Except for the connector object class and container classes, most objects must have a property sheet with one or more pages. If your connector contains container classes with inherit objects, you must create a property sheet with property pages for the container. The GUI framework provides a predefined property sheet for connector classes that you can use.

The following provides important information about property pages:

- Property page titles represent the tab title in the property sheet.
- If you want to include an Inclusion property page or a Shared COS (common object) page, you must also add the appropriate resources to your code by including them in your resource file. To do this, select Resource Includes from the View menu, and enter the following lines in the read-only symbol directive field:

```
#include "afxres.h"
#ifdef APSTUDIO_INVOKED
#include "../../include/GUI/CAInclusionPage.rc"
#include "../../include/GUI/CosShare.rc"
#endif
```

- Policy property pages contain capability attributes that should appear in bold type. To make the static text boldface, use the predefined `CFont` member variable `CAPropertyPage`, `fBoldFont`. You can access this member if your policy property pages are derived from `CAPropertyPage`.

To apply the bold font to a control on your property page, call the `SetFont` function for that control in your `OnInitDialog` function (or another initialization function) as follows:

```
CStaticControlMember.SetFont(&fBoldFont)
```

SetFont is part of the CWnd base class, so all controls can call this function.

Note: The bold text is longer than the normal text that you see in the design view of Visual Studio. Make sure that you leave enough room for it.

If you use the same property pages for your accounts as your policies, you must determine the object type before boldfacing the controls. This can be accomplished in the constructor by assigning a value to an int member called `isPolicySheet` and using the following code example:

```
isPolicySheet = (wcsncmp(propertySheet->pszClassName, your_policy_classname)
== 0);
```

where *your_policy_classname* is the attribute name (*not* the naming attribute) of your Policy class.

and your code for bolding the controls tests the int value:

```
if (isPolicySheet)
    CStaticControlMember.SetFont(&fBoldFont)
```

This prevents the control text from being made bold if this is an account property page.

Property Sheets and Property Pages

Each property page lets you manipulate a set of properties or attributes. These properties are subclasses from the `CAPropertyPage` class defined in `CAPropertyPage.h`. Sometimes, you may want to base your class on the `CADomainInclusionPage`, `CASelectionPage`, `CosGenericStatPage`, or `CosPolicyPage` pages to gain additional functionality. These pages are all based on `CAPropertyPage`.

The GUI framework must have the relevant information for loading the correct property pages for each property sheet.

Create the C++ Code for a Property Sheet

Execute the following procedure to create the code for a property sheet.

To create the C++ code for a property sheet

1. Specify `CAPropertySheet`, or one of its derived classes, as the base class for your property sheet.

Note: The `CAPropertySheet` is derived from the MFC object `CPropertySheet`.

2. Supply a constructor for your property sheet.

Your constructor must invoke the base class constructor for `CAPropertySheet`, and call the `propertyPages`, `AddTail`(*page_object*) function for each page in the property sheet.

page_object

Pointer to an instantiation of the C++ class representing the property page.

3. For existing objects with a false `isNewObject` value, add `CosStatisticsPage` to your property sheet.

Note: An existing object is identified when its `isNewObject` member variable has been set to `False` by the framework.

4. Perform a `loadPages()` call.

The following example illustrates the property sheet code for the SDK Namespace account property sheet.

```
#include "stdafx.h"
#include "unitypes.h"
#include <afxpriv.h>
#include "etacldap.h"
#include "etaclass.h"
#include "caedit.h"
#include "CAPropertySheet.h"
#include "CAPropertyPage.h"
#include "CAInclusionPage.h"
#include "CosStatisticsPage.h"

#include "CASDKGUI.h"
#include "CASDKCLS.h"
#include "CASDKDirectorySheet.h"
#include "CASDKDirectoryPropertyPage.h"
```

```
////////////////////////////////////
// Construction/Destruction
////////////////////////////////////

IMPLEMENT_DYNAMIC(CASDKDirectorySheet, CAPropertySheet)

CASDKAccountSheet::CASDKAccountSheet(
    CWnd*          pParentWnd.
    PWCHAR_T      pszClassName.
    PWCHAR_T      pszObjectDN.
    HWND          hGui.
    ETA_HANDLE     hRepository.
    BOOL           isNewObject.
    BOOL           isModal.
    BOOL           confirmDelete.
    BOOL           globalSelects
)
{
    // Invoke the superclass constructor
    :CAPropertySheet (
        pParentWnd.
        pszClassName.
        pszObjectDN.
        hGui.
        hRepository.
        isNewObject.
        isModal.
        confirmDelete.
        globalSelects
    )
{
    // One property page is sufficient for the basic SDKAccount attributes
    propertyPages->AddTail (new CASDKAccountPropertyPage (this));

    /*
    || Add the inclusion pages to represent the groups this user belongs to
    */
    propertyPages->AddTail (new CAInclusionPage (this.
                                                SRD_ETASDKACCOUNT.
                                                pszObjectDN.
                                                SRD_ETASDKGROUP.
                                                IDS_GROUPS));

    // Add the framework
    // supplied statistical info page.
    if( !isNewObject )
        propertyPages->AddTail (new CosStatisticsPage (this));

    loadPages();
}
```


Property Page Code

The property page code retrieves the property values from the DIT and displays them in a property page. When an administrator changes the values of some of these properties, the code must retrieve these values from the page, and store the modified values back in the DIT.

Note: The property values are transferred from the property page to Manager using a data structure called the SDS buffer.

For the following standard MFC controls, the GUI framework provides functions to transfer values directly from the SDS buffer to the corresponding MFC control or object:

- CEdit
- CButton
- CSpinButtonCtrl
- CComboBox
- CDateTimeCtrl
- CStringArray

Use the `setData( )` function to transfer data from a property page stored in the SDS buffer to any one of the standard controls listed above. Use the function `setSdsData( )` to transfer data from the control to the outbound SDS buffer.

For other controls, you must first call the GUI framework functions to manually retrieve data from the SDS buffer, and then call the appropriate MFC functions to populate the corresponding controls. The property pages will usually specify one of the following calls to retrieve attribute values directly:

```
int foo = propertySheet->sdsOriginal->getInt( PROPERTY_NAME) PWCHAR_T somePtr =  
propertySheet->sdsOriginal->getString(PROPERTY_NAME)
```

These examples assume that “propertySheet” is a pointer to the property sheet (this is the existing definition in `CAPropertyPage`, which is available to all derived classes), and that `PROPERTY_NAME` is the name of the property that you want to retrieve.

You can store raw data using one of the following functions, which are available to all property pages:

```
void setSdsData(PWCHAR_T pszProperty, const PWCHAR_T pszdata);
```

```
void setSdsData(PWCHAR_T pszProperty, int iData);
```

Your property pages must supply the following member functions:

- `void setData( )`--Initializes the property page when it first appears and on any future appearances. It retrieves the attribute values from the GUI framework and initializes the corresponding MFC controls.
- `int apply( )`--Sends updated values back to the GUI framework when a user makes changes.
- `CWnd*setErrorField(PWCHAR_T errorField)`--Sets the focus on a field when the GUI framework detects an error.

More information:

[GUI Framework APIs](#) (see page 273)

For more information about the property pages, see the following files in the *PSDKHOME/Samples/SDK/GUI* directory:

- `CAPropertyPage.h`--Contains the `CAPropertyPage` class definition (in *PSDKHOME/include/GUI*).
- `CASdsBuffer.h`--Provides information on the SDS buffer class that is used to contain the data from the Provisioning Directory (in *PSDKHOME/include/GUI*).
- `CA*Page.cpp`--Displays sample property pages that you can use.

Code Examples

SDKAccountPropertyPage.h Code Example

The following sample code is from the SDKAccountPropertyPage header file.

Note: Each attribute has a separate declaration that appears on the property page.

```
////////////////////////////////////
// CASDKAccountPropertyPage dialog

class CASDKAccountPropertyPage : public CPropertyPage
{
    DECLARE_DYNAMIC(CASDKAccountPropertyPage)

// Construction
public:
    CASDKAccountPropertyPage(CPropertySheet * propertySheet);
    CASDKAccountPropertyPage(CWnd* pParent = NULL);
    ~CASDKAccountPropertyPage();

// Dialog Data
   //{{AFX_DATA(CASDKAccountPropertyPage)
    enum { IDD = IDD_ACCOUNT_PROPERTIES };
    CEdit    m_edtCity;
    CEdit    m_edtEmployeeId;
    CSpinButtonCtrl m_spinEmployeeId;
    CEdit    m_edtPasswd;
    CEdit    m_edtConfPasswd;
    CButton  m_chkEncrypt;
    //}}AFX_DATA

// Overrides
    // ClassWizard generate virtual function overrides
    //{{AFX_VIRTUAL(CASDKAccountPropertyPage)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL
```

```
// Implementation
protected:
    // Generated message map functions
   //{{AFX_MSG(CASDKAccountPropertyPage)
    afx_msg void OnChangeAccCity();
    afx_msg void OnChangeAccEmployeeid();
    afx_msg void OnChangeAccPassword();
    afx_msg void OnChangeAccConfpass();
    afx_msg void OnAccEncryptChk();
   //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
    /* set or reset data; called when initializing dialog or the user
       pressed reset button */
    void setData();
    /* User pressed the apply/OK button */
    int apply();
    /* Called when an error has occurred; used to determine where focus
       should be set */
    CWnd * setErrorField (PWCHAR_T errorField);

};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before
// the previous line.

#endif //
!defined(AFX_CASDKACCOUNTPROPERTYPAGE_H__C7775BA9_0C68_41E7_96D9_9C5BBFCCECEA__IN
CLUDED_)
```

SDKAccountPage.cpp Code Example

The following sample code is from the SDKAccountPropertyPage.cpp file.

```
////////////////////////////////////  
// CASDKAccountPropertyPage property page  
  
IMPLEMENT_DYNAMIC(CASDKAccountPropertyPage, CAPropertyPage)  
  
CASDKAccountPropertyPage::CASDKAccountPropertyPage(  
    CAPropertySheet * propertySheet) :  
    CAPropertyPage(propertySheet, CASDKAccountPropertyPage::IDD)  
{  
   //{{AFX_DATA_INIT(CASDKAccountPropertyPage)  
   //}}AFX_DATA_INIT  
}  
  
CASDKAccountPropertyPage::CASDKAccountPropertyPage(CWnd* pParent /*=NULL*/)  
    : CAPropertyPage()  
{  
}  
  
CASDKAccountPropertyPage::~CASDKAccountPropertyPage()  
{  
}  
  
void CASDKAccountPropertyPage::DoDataExchange(CDataExchange* pDX)  
{  
    CAPropertyPage::DoDataExchange(pDX);  
    {{{AFX_DATA_MAP(CASDKAccountPropertyPage)  
    DDX_Control(pDX, IDC_ACC_CITY, m_edtCity);  
    DDX_Control(pDX, IDC_ACC_EMPLOYEEID_SPIN, m_spinEmployeeId);  
    DDX_Control(pDX, IDC_ACC_EMPLOYEEID, m_edtEmployeeId);  
    DDX_Control(pDX, IDC_ACC_PASSWORD, m_edtPasswd);  
    DDX_Control(pDX, IDC_ACC_CONFPASS, m_edtConfPasswd);  
    DDX_Control(pDX, IDC_ACC_ENCRYPT_CHK, m_chkEncrypt);  
    }}}AFX_DATA_MAP  
}
```

```
BEGIN_MESSAGE_MAP(CASDKAccountPropertyPage, CAPropertyPage)
//{{AFX_MSG_MAP(CASDKAccountPropertyPage)
    ON_EN_CHANGE(IDC_ACC_CITY, OnChangeAccCity)
    ON_EN_CHANGE(IDC_ACC_EMPLOYEEID, OnChangeAccEmployeeid)
    ON_EN_CHANGE(IDC_ACC_PASSWORD, OnChangeAccPassword)
    ON_EN_CHANGE(IDC_ACC_CONFPASS, OnChangeAccConfpass)
    ON_BN_CLICKED(IDC_ACC_ENCRYPT_CHK, OnAccEncryptChk)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////
// CASDKAccountPropertyPage message handlers

void CASDKAccountPropertyPage::OnChangeAccCity()
{
    // TODO: If this is a RICHEDIT control, the control will not
    // send this notification unless you override the
    CAPropertyPage::OnInitDialog()
    // function and call CRichEditCtrl().SetEventMask()
    // with the ENM_CHANGE flag ORed into the mask.

    // TODO: Add your control notification handler code here
    CAPropertyPage::OnChange();
}

void CASDKAccountPropertyPage::OnChangeAccEmployeeid()
{
    // TODO: If this is a RICHEDIT control, the control will not
    // send this notification unless you override the
    CAPropertyPage::OnInitDialog()
    // function and call CRichEditCtrl().SetEventMask()
    // with the ENM_CHANGE flag ORed into the mask.

    // TODO: Add your control notification handler code here
    CAPropertyPage::OnChange();
}

void CASDKAccountPropertyPage::OnChangeAccPassword()
{
    // TODO: If this is a RICHEDIT control, the control will not
    // send this notification unless you override the
    CAPropertyPage::OnInitDialog()
    // function and call CRichEditCtrl().SetEventMask()
    // with the ENM_CHANGE flag ORed into the mask.

    // TODO: Add your control notification handler code here
    CAPropertyPage::OnChange();
}
```

```
}

void CASDKAccountPropertyPage::OnChangeAccConfpass()
{
    // TODO: If this is a RICHEDIT control, the control will not
    // send this notification unless you override the
    CAPropertyPage::OnInitDialog()
    // function and call CRichEditCtrl().SetEventMask()
    // with the ENM_CHANGE flag ORed into the mask.

    // TODO: Add your control notification handler code here
    CAPropertyPage::OnChange();
}

void CASDKAccountPropertyPage::OnAccEncryptChk()
{
    // TODO: Add your control notification handler code here
    CAPropertyPage::OnChange();
}

/*
|| initialize all controls
||
|| This routine is called when the dialog is initialized, and when
|| the user presses the reset button. This routine gets the data from
|| the original SDS buffer. The setData() members of the CAPropertyPage
|| perform this function, as well as any control initialization (entry field
|| maximum lengths, spin button ranges, and so on)
||
*/
void CASDKAccountPropertyPage::setData(){
    if (!bInit){
        // Subclass the edit controls first time through
        /*
        m_edtCity.SubclassDlgItem(IDC_ACC_CITY, this);
        m_edtEmployeeId.SubclassDlgItem(IDC_ACC_EMPLOYEEID, this);
        */
    }
    CAPropertyPage::setData (SRD_ETASDKACC_CITY, &m_edtCity);
    CAPropertyPage::setData (SRD_ETASDKACC_EMPLOYEEID, &m_edtEmployeeId,
true);
    CAPropertyPage::setData (SRD_ETASDKACC_PASSWORD, &m_edtPasswd);
    CAPropertyPage::setData (SRD_ETASDKACC_PASSWORD, &m_edtConfPasswd);
    CAPropertyPage::setData (SRD_ETASDKACC_ENCRYPT, &m_chkEncrypt);

    /*
```

```
        || These properties cannot be modified for an existing object
        */
    m_spinEmployeeId.SetRange(0, 99);
}

/*
|| Apply/OK pressed
||
|| set any changed data in the sdsNew buffer in the property sheet.
|| The setSdsData() members in the CAPropertyPage superclass will
|| check to see if the property has changed, and add it to the SDS
|| buffer. If the setSdsData() member returns TRUE, the attribute
|| has changed, so the return code is set to PAGE_MODIFIED
*/
int CASDKAccountPropertyPage::apply ()
{
    int rc = PAGE_UNMODIFIED;

    if (bInit){
        if(!UpdateData(TRUE)){ // DoDataExchange
            return PAGE_ERROR;
        }
        if (setSdsData (SRD_ETASDKACC_CITY, &m_edtCity)){
            rc = PAGE_MODIFIED;
        }
        if (setSdsData (SRD_ETASDKACC_EMPLOYEEID, &m_edtEmployeeId,
true)){
            rc = PAGE_MODIFIED;
        }
        // extra handling for password property since we do not store
real value in control content
        CString OldPwd, NewPwd;
        CString OldCfmPwd, NewCfmPwd;
        OldPwd = propertySheet->sdsOriginal-
>getString(SRD_ETASDKACC_PASSWORD);
        m_edtPasswd.GetWindowText(NewPwd);

        OldCfmPwd = propertySheet->sdsOriginal-
>getString(SRD_ETASDKACC_PASSWORD);
        m_edtConfPasswd.GetWindowText(NewCfmPwd);

        if (OldPwd != NewPwd || OldCfmPwd != NewCfmPwd ) {
            // verify
            if (NewPwd == NewCfmPwd) {
                if (propertySheet->isNewObject()) {
                    setSdsData (SRD_ETASDKACC_PASSWORD,
NewPwd.GetBuffer (NewPwd.GetLength()));
                    rc = PAGE_MODIFIED;
                }
            }
        }
    }
}
```



```
        else {
            if (setSdsData(SRD_ETASDKACC_PASSWORD,
&m_edtPasswd)){
                rc = PAGE_MODIFIED;
            }
        }
    }
    else {
        // 3 : ID_ERR_PASSWORD_MUST_MATCH = "The confirm
password must match new password"
        CAUmsgf(CAM_REQPARMS.
                CAIMSG_SDKDLL.
                ID_ERR_PASSWORD_MUST_MATCH.
                CAIMSG_SEV_ERROR.
                CAM_PRINTW.
                NULL);
        m_edtConfPasswd.SetSel(0, -1);
        propertySheet->SetActivePage(this);
        m_edtConfPasswd.SetFocus();
        return PAGE_ERROR;
    }
}
if (setSdsData(SRD_ETASDKACC_ENCRYPT, &m_chkEncrypt)){
    rc = PAGE_MODIFIED;
}
}

return rc;
}

/*
|| overrideable to catch errors and set focus appropriately
|| the errorfield will be the field that is determined to be in error
|| if the field is on this page, we will return the control that will
|| gain focus on return, along with an appropriate error message
*/
CWnd * CASDKAccountPropertyPage::setErrorField (PWCHAR_T errorField)
{
    if (wcscmp (errorField, SRD_ETASDKACC_CITY) == 0){
        return &m_edtCity;
    }
    else if (wcscmp (errorField, SRD_ETASDKACC_EMPLOYEEID) == 0){
        return &m_edtEmployeeId;
    }
    else if (wcscmp (errorField, SRD_ETASDKACC_PASSWORD) == 0){
        return &m_edtPasswd;
    }
    else if (wcscmp (errorField, SRD_ETASDKACC_ENCRYPT) == 0){
        return &m_chkEncrypt;
    }
}
```

```
    }  
  
    return NULL;  
}
```

Message Map Information

The following message map lists the functions that are executed when a message is received. For messages that indicate when a field value changes, the `OnChange( )` function in the `CAPropertyPage` must be called to enable the Apply button. This button is initially grayed out. You can specify it here as a function or you can use a handler of your own by calling the `OnChange( )` function as shown in the following message map:

```
BEGIN_MESSAGE_MAP(CASDKAccountPropertyPage, CAPropertyPage)  
    //{AFX_MSG_MAP(CASDKAccountPropertyPage)  
    ON_EN_CHANGE(IDC_ACC_LOGINNAME,      OnChangeAccLoginname)  
    ON_EN_CHANGE(IDC_ACC_CITY,           OnChangeAccCity)  
    ON_EN_CHANGE(IDC_ACC_EMPLOYEEID,     OnChangeAccEmployeeid)  
    ON_EN_CHANGE(IDC_ACC_PASSWORD,       OnChangeAccPassword)  
    ON_EN_CHANGE(IDC_ACC_CONFPASS,       OnChangeAccConfpass)  
    ON_CBN_SELCHANGE(IDC_ACC_STATUS,     OnSelchangeAccStatus)  
    ON_BN_CLICKED(IDC_ACC_ENCRYPT_CHK,    OnAccEncryptChk)  
    //}}AFX_MSG_MAP  
END_MESSAGE_MAP()  
  
/////////////////////////////////////  
// CASDKAccountPropertyPage message handlers  
  
void CASDKAccountPropertyPage::OnChangeAccLoginname()  
  
{  
    // TODO: If this is a RICHEDIT control, the control will not  
    // send this notification unless you override the CAPropertyPage::OnInitDialog()  
    // function and call CRichEditCtrl().SetEventMask()  
    // with the ENM_CHANGE flag ORed into the mask.  
  
    // TODO: Add your control notification handler code here  
    CAPropertyPage::OnChange();  
}
```

```
void CASDKAccountPropertyPage::OnChangeAccCity()
{
    // TODO: If this is a RICHEDIT control, the control will not
    // send this notification unless you override the
    // CAPropertyPage::OnInitDialog()
    // function and call CRichEditCtrl().SetEventMask()
    // with the ENM_CHANGE flag ORed into the mask.

    // TODO: Add your control notification handler code here
    CAPropertyPage::OnChange();
}

void CASDKAccountPropertyPage::OnChangeAccEmployeeid()
{
    // TODO: If this is a RICHEDIT control, the control will not
    // send this notification unless you override the
    // CAPropertyPage::OnInitDialog()
    // function and call CRichEditCtrl().SetEventMask()
    // with the ENM_CHANGE flag ORed into the mask.

    // TODO: Add your control notification handler code here
    CAPropertyPage::OnChange();
}

void CASDKAccountPropertyPage::OnChangeAccPassword()
{
    // TODO: If this is a RICHEDIT control, the control will not
    // send this notification unless you override the
    // CAPropertyPage::OnInitDialog()
    // function and call CRichEditCtrl().SetEventMask()
    // with the ENM_CHANGE flag ORed into the mask.

    // TODO: Add your control notification handler code here
    CAPropertyPage::OnChange();
}
```

```
void CASDKAccountPropertyPage::OnChangeAccConfpass()
{
    // TODO: If this is a RICHEDIT control, the control will not
    // send this notification unless you override the
    // CAPropertyPage::OnInitDialog()
    // function and call CRichEditCtrl().SetEventMask()
    // with the ENM_CHANGE flag ORed into the mask.

    // TODO: Add your control notification handler code here
    CAPropertyPage::OnChange();
}

void CASDKAccountPropertyPage::OnSelchangeAccStatus()
{
    // TODO: Add your control notification handler code here
    CAPropertyPage::OnChange();
}

void CASDKAccountPropertyPage::OnAccEncryptChk()
{
    // TODO: Add your control notification handler code here
    CAPropertyPage::OnChange();
}
```

setData() Method Information

The following setData() method initializes the display. For first-time initializations and re-initializations, when an administrator clicks Reset, place the initialization code in the !bInit conditional clause. This is useful for linking the controls with their corresponding data elements. In this example, it is not required because the MFC-supplied DDX function accomplishes that. User code contained in the MFC OnInitDialog() function could be placed here instead.

Note: The CAPropertyPage::setData() method called for each control must transfer data from the specified property to the corresponding dialog control. The fields can be set for a new object, but cannot be changed for an existing object. The fields are disabled through the call to EnableWindow().

```
/*
|| initialize all controls
||
|| This routine is called when the dialog is initialized, and when
|| the user presses the reset button. This routine gets the data from
|| the original SDS buffer. The setData() members of the CAPropertyPage
|| perform this function, as well as any control initialization (entry field
|| maximum lengths, spin button ranges, and so on)
||
*/
void CASDKAccountPropertyPage::setData(){

    if (!bInit){
        // Link the data elements with the corresponding MFC control
        /*
        m_edtLoginName.SubclassDlgItem(IDC_ACC_LOGINNAME, this);
        m_edtCity.SubclassDlgItem(IDC_ACC_CITY, this);
        m_edtEmployeeId.SubclassDlgItem(IDC_ACC_EMPLOYEEID, this);
        */
    }

    CAPropertyPage::setData (SRD_ETASDKACC_LOGINNAME,      &m_edtLoginName);
    CAPropertyPage::setData (SRD_ETASDKACC_CITY,           &m_edtCity);
    CAPropertyPage::setData (SRD_ETASDKACC_EMPLOYEEID,     &m_edtEmployeeId,
true);
    CAPropertyPage::setData (SRD_ETASDKACC_ENCRYPT,         &m_chkEncrypt);
    CAPropertyPage::setData (SRD_ETASDKACC_PASSWORD,       &m_edtPasswd);
    CAPropertyPage::setData (SRD_ETASDKACC_PASSWORD,       &m_edtConfPasswd);
    CAPropertyPage::setData (SRD_ETASDKACC_STATUS,         &m_comboStatus);

    /*
    || These properties cannot be modified for an existing object
    */
    if (wcscmp(propertySheet->pszClassName, SRD_ETASDKACCOUNT) == 0){
        m_edtLoginName.EnableWindow (propertySheet->isNewObject());
    }
    m_spinEmployeeId.SetRange(0, 99);
}
```

apply() Method Information

The apply() method is called when a user clicks the Apply or OK buttons. Note its use of the setSdsData() routine to copy the edit box data field into the SDS buffer. If the field was changed, setSdsData() returns TRUE, in which case the routine sets the PAGE_MODIFIED flag. This notifies the GUI framework that the data has been changed and the SDS buffer must be rewritten. The CAUmsgf() routine is called to send a message to the Unicenter TNG console or Log Viewer.

```
/*
|| Apply/OK pressed
||
|| set any changed data in the sdsNew buffer in the property sheet.
|| The setSdsData() members in the CAPropertyPage superclass will
|| check to see if the property has changed, and add it to the SDS
|| buffer. If the setSdsData() member returns TRUE, the attribute
|| has changed, so the return code is set to PAGE_MODIFIED
*/
int CASDKAccountPropertyPage::apply()
{
    int rc = PAGE_UNMODIFIED;

    if (bInit){
        if(!UpdateData(TRUE)){ // DoDataExchange
            return PAGE_ERROR;
        }

        CString sTmpAcc;
        m_edtLoginName.GetWindowText(sTmpAcc);
        if( sTmpAcc.GetLength() == 0 ){
            // 2 : ID_ERR_LOGINNAME_MANDATORY = "Login Name is mandatory"
            CAUmsgf(CAM_REQPARMS.
                    CAIMSG_SDKDLL.
                    ID_ERR_LOGINNAME_MANDATORY.
                    CAIMSG_SEV_ERROR.
                    CAM_PRINTW.
                    NULL);
            m_edtLoginName.SetFocus();
            m_edtLoginName.SetSel(0, -1);
            return PAGE_ERROR;
        }
    }
```

```
if (setSdsData (SRD_ETASDKACC_LOGINNAME, &m_edtLoginName)){
    rc = PAGE_MODIFIED;
}
if (setSdsData (SRD_ETASDKACC_CITY, &m_edtCity)){
    rc = PAGE_MODIFIED;
}
if (setSdsData (SRD_ETASDKACC_EMPLOYEEID, &m_edtEmployeeId, true)){
    rc = PAGE_MODIFIED;
}
// extra handling for password property since we do not store the
// real value in control content
CString OldPwd, NewPwd;
CString OldCfmPwd, NewCfmPwd;
OldPwd = propertySheet->sdsOriginal->getString(SRD_ETASDKACC_PASSWORD);
m_edtPasswd.GetWindowText(NewPwd);

OldCfmPwd = propertySheet->sdsOriginal-
>getString(SRD_ETASDKACC_PASSWORD);
m_edtConfPasswd.GetWindowText(NewCfmPwd);

if (OldPwd != NewPwd || OldCfmPwd != NewCfmPwd ) {
    // verify
    if (NewPwd == NewCfmPwd) {
        if (propertySheet->isNewObject()) {
            setSdsData (SRD_ETASDKACC_PASSWORD,
                NewPwd.GetBuffer (NewPwd.GetLength()));
            rc = PAGE_MODIFIED;
        }
        else {
            if (setSdsData(SRD_ETASDKACC_PASSWORD, &m_edtPasswd)){
                rc = PAGE_MODIFIED;
            }
        }
    }
    else {
        // 3 : ID_ERR_PASSWORD_MUST_MATCH = "The confirm password must match new
        password"
        CAUmsgf(CAM_REQPARMS.
            CAIMSG_SDKDLL.
            ID_ERR_PASSWORD_MUST_MATCH.
            CAIMSG_SEV_ERROR.
            CAM_PRINTW.
            NULL);
        m_edtConfPasswd.SetFocus();
        m_edtConfPasswd.SetSel(0, -1);
        return PAGE_ERROR;
    }
}
```

```
        if (setSdsData(SRD_ETASDKACC_ENCRYPT, &m_chkEncrypt)){
            rc = PAGE_MODIFIED;
        }

        if (setSdsData (SRD_ETASDKACC_STATUS, &m_comboStatus)){
            rc = PAGE_MODIFIED;
        }
    }

    return rc;
}

/*
|| overrideable to catch errors and set focus appropriately
|| the errorfield will be the field that is determined to be in error
|| if the field is on this page, we will return the control that will
|| gain focus on return, along with an appropriate error message
*/
CWnd * CASDKAccountPropertyPage::setErrorField (PWCHAR_T errorField)
{
    if (wcscmp (errorField, SRD_ETASDKACC_LOGINNAME) == 0){
        return &m_edtLoginName;
    }
    else if (wcscmp (errorField, SRD_ETASDKACC_CITY) == 0){
        return &m_edtCity;
    }
    else if (wcscmp (errorField, SRD_ETASDKACC_EMPLOYEEID) == 0){
        return &m_edtEmployeeId;
    }
    else if (wcscmp (errorField, SRD_ETASDKACC_ENCRYPT) == 0){
        return &m_chkEncrypt;
    }
    else if (wcscmp (errorField, SRD_ETASDKACC_PASSWORD) == 0){
        return &m_edtPasswd;
    }
    else if (wcscmp (errorField, SRD_ETASDKACC_STATUS) == 0){
        return &m_comboStatus;
    }

    return NULL;
}
```


Building and Linking the GUI Plug-in

To build and link the GUI plug-in, use the makefile supplied with the SDK as a template, substituting the names of your files for those listed in the SDK.

You might also need to adjust the following variables:

FIRSTINC and LASTINC

Add any include directories to the beginning or ending of the search path.

FIRSTLIB and LASTLIB

Add any required libraries to the beginning or ending of the library path.

Adding Objects to the Provisioning Directory

You must add your connector's predefined objects to the Provisioning Directory.

To do this, you may use the Populate directory which is included with the SDK sample source files. Use the `sdkpop.h` file as a template for your own `.h` file. This file should be included in the corresponding `.c` or `.cpp` file.

Linking and Providing Help Files

To link your help file with the GUI, you must add a line similar to the following for each of your classes in the parser table:

```
Win32help=sdk.hlp
```

```
sdk.hlp
```

Unique name of the help file for your connector.

Note: This name is important for the help mechanism. It is used in the help map file, named `map sdk.txt`, which contains the links between the dialog and property page IDs in the GUI plug-in and the corresponding HTML files containing the help.

Install and Test the GUI Plug-in

Before you install and test the GUI plug-in, verify that you have done the following:

- For each dialog, you have coded a property sheet. Each property sheet can contain any number of property pages. The property page class is derived from the `CAPropertyPage` class.
- For each dialog, you have defined three member functions that will override the parent member functions. These functions, `setData( )`, `apply( )`, and `setErrorField( )`, handle the interaction between the dialog and the data from the directory.
- You have defined your `Directory` class and your tree hierarchy in the parser table.

Chapter 10: Building an Agent Plug-In

The Provisioning Server provides a superagent framework for third-party agent plug-in developers. Each agent plug-in must manage one endpoint type. For example, an Oracle agent plug-in manages the users and roles in an Oracle endpoint type.

The following list provides the major tasks that you must perform to code and build an agent plug-in.

1. Include required files.
2. Derive a class from the `DMODirectoryEntry` class for each DIT node.
3. Implement the functions for each object class.
4. Implement the factory functions.
5. Initialize the agent plug-in.
6. Build and link the agent plug-in.
7. Install and test the agent plug-in.

Note: You must build a GUI plug-in for each agent plug-in that you build.

This section contains the following topics:

[Required Files to be Included](#) (see page 163)
[Deriving a Class for Each Node](#) (see page 164)
[Implement the Functions for an Object Class](#) (see page 165)
[Implement the Factory Functions](#) (see page 166)
[Initialize the Agent Plug-In](#) (see page 167)
[Build an Agent Plug-In](#) (see page 171)
[Test the Agent Plug-in DLL](#) (see page 174)

Required Files to be Included

The required `#include` files are the following:

File Name	Contents
<code>DMOAgent.H</code>	Basic <code>#include</code> file for an agent plug-in.
<code>DMODirectoryEntry.H</code>	Code for the <code>DMODirectoryEntry</code> base class.
<code>DMO_LDAP_Entry.H</code>	Class that represents a directory entry.
<code>DMO_LDAP_Request.H</code>	Class that represents an LDAP request.

Deriving a Class for Each Node

For each node in your DIT, you must derive a unique LDAP object class name from the `DMODirectoryEntry` class.

More information:

[Understanding Predefined Object Classes](#) (see page 69)

[Derive a Class for STATIC DITs](#) (see page 164)

[Derive a Class for Dynamic DITs](#) (see page 165)

Derive a Class for Static DITs

Use the following constructor code for static DITs:

```
CSampleStaticInternalNode::CSampleStaticInternalNode(  
    UTF8      *pszObjectClass,  
    UTF8      *pszRdnType,  
    UTF8      *pszRdnValue  
    ) : DMODirectoryEntry(pszObjectClass, pszRdnType, pszRdnValue)
```

CSampleStaticInternalNode is the name of the object class you are deriving.

Derive a Class for Dynamic DITs

Use the following constructor code for dynamic DITs. Ensure that you set the class variable mode to indicate a dynamic node.

The following code shows an example of a constructor for a dynamic node:

```
CSampleDynamicInternalNode::CSampleDynamicInternalNode(
    UTF8      *pszObjectClass,
    UTF8      *pszRdnType,
    UTF8      *pszRdnValue
) : DMODirectoryEntry(pszObjectClass, pszRdnType, pszRdnValue)
{
    int default_mode = getMode();
    int dynamic_mode = default_mode | (DE_MODE_DYNAMIC | DE_MODE_PENDING);
    setMode(dynamic_mode);
}
```

CSampleDynamicInternalNode is the name of the object class you are deriving.

Implement the Functions for an Object Class

After deriving the object class names for the nodes in your DIT, you must add LDAP functions, such as ADD, SEARCH, and MODIFY to each object class. The Provisioning Server uses these functions when managing the connectors.

Typically, the following node types implement the following functions:

Node Type	Functions
Internal nodes	DEinit, DEsearch, DEmodify
Leaf nodes	DEinit, DEsearch, DEadd, DEdelete, DEmodify

For example, to perform a delete on a UNIX account, the function is coded as follows:

```
int
CSampleInternalNode::DEdelete(
    DistinguishedName    *pTargetDN,
    DMOMessage           *pStatusMsg
)
{
```

```
int    rc    =    LDAP_SUCCESS;

/*
|| Add any agent plug-in specific code here that need the node to
|| still be attached to the DIT.
*/

/*
|| Detach the directory and its descendent nodes from the main DIT.
*/
rc = detach();

/*
|| Add any agent plug-in specific code here that must be done after
|| the node is detached.
*/

return rc;
}
```

CSampleInternalNode is the name of the object class you are deriving.

More information:

[Superagent Framework APIs](#) (see page 289)

Implement the Factory Functions

In addition to the LDAP functions, you must also implement the necessary factory functions. For each class you derive from *DMODirectoryEntry*, you must provide a factory function.

To declare a factory function, use the following code:

```
DMODirectoryEntry*
FactoryFunc(char *objectClass, char *rdn_type, char *rdn_value);
```

FactoryFunc is the name of the factory function and *objectClass* is the name of the object class.

The following is an example of a factory function:

```
DMODirectoryEntry *
sampleAccountFolder_DEFactory(
    UTF8      *pszObjectClass,
    UTF8      *pszRdnType,
    UTF8      *pszRdnValue
)
{
    return new CSampleAccountFolder(pszObjectClass, pszRdnType, pszRdnValue);
}
```

Initialize the Agent Plug-In

All agent plug-ins must export the function `init_agent( )`:

```
int __declspec(dllexport) init_agent(DERepository **ppRep)
```

When the superagent loads the plug-in DLL, it calls this function to initialize the agent plug-in. As part of the initialization, the plug-in must register a set of factory functions. The `init_agent( )` function registers the factory functions by returning a pointer to an array of `DERepository`, which contains the addresses of the factory functions. The array must terminate with a `DERepository` record in which the `objectclass` field is `NULL`.

The following is an example:

```
DERepository EntryPointTable[] = {
    {NULL, "sampleNamespace", sampleNamespace_DEFactory},
    {NULL, "sampleDirectory", sampleDirectory_DEFactory},
    {NULL, "sampleAccountFolder", sampleAccountFolder_DEFactory},
    {NULL, "sampleAccount", sampleAccount_DEFactory},
    {NULL, NULL, NULL} // Terminate the array
};
```

Given the array above, the `init_agent( )` function appears similar to the following:

```
int __declspec(dllexport)
init_agent(DERepository **ppRep)
{
    int rc = 0;

    if (ppRep == NULL) {
        rc = -1;
        goto exit;
    }
    *ppRep = EntryPointTable;

    /*
    || Do license check here. If the license check fail, return
    || DMO_AGENT_INVALID_LICENSE.
    */
    /*
    || Do any other initialization here.
    */

exit:
    return rc;
}
```

In addition to the `init_agent( )` function, each agent plug-in can also export a shutdown function:

```
int __declspec(dllexport) shutdown_agent()
```

Enable Thread Support of Agent Plug-Ins

The superagent is a back end to the SLAPD front-end, which is multi-threaded. Multiple simultaneous requests to the superagent are handled by different threads. When the requests reach the superagent at the same time, the superagent may or may not dispatch multiple requests to the same agent plug-in at the same time, depending on how agent plug-ins register themselves to the superagent. Before enabling the threading-support for an agent plug-in, the agent plug-in must be validated that it is thread-safe. In addition to ensuring that the agent plug-in code is thread-safe, the agent plug-in also needs to register itself as thread-safe to the superagent.

Each C++ agent plug-in code that is to handle one object class in managed systems is derived from the `DMODirectoryEntry` class. The C++ class derived, without specifying any thread-safety, is considered NOT thread-safe. As such, the superagent will serialize accesses to the agent plug-in.

For example, the following SDK account class is declared to the superagent as NON-thread-safe:

```
CSDKAccount::CSDKAccount(  
    UTF8 *pszObjectClass,  
    UTF8 *pszRdnType,  
    UTF8 *pszRdnValue  
) : DMOldDirectoryEntry(  
    pszObjectClass,  
    pszRdnType,  
    pszRdnValue)  
{  
    ...  
}
```

As another example, the following code segment declares that the SDK account class is thread-safe. As such, the superagent permits simultaneous accesses to the agent plug-in code.

```
CSDKAccount::CSDKAccount(  
    UTF8 *pszObjectClass,  
    UTF8 *pszRdnType,  
    UTF8 *pszRdnValue  
) : DMOldDirectoryEntry(  
    pszObjectClass,  
    pszRdnType,  
    pszRdnValue,  
    true) // indicating that the class is thread-safe  
{  
    ...  
}
```

Note: Not only does the “account” class constructor need to be updated, but also the DIT parent class, such as “account container”, “directory”, and “namespace.”

Enable Parallel Search in an Agent Plug-In

For any class that is a leaf node (for example, account or group), the class must specify the newly defined mode, `DE_MODE_MULTI_SEARCH`. The following is an example:

```
SDKAccount::SDKAccount()
{
    // Allow multiple searches on the account leaf-node at the same time
    // NOTE: we OR the DE_MODE_MULTI_SEARCH bit with the current default bit

    setMode(getMode() | DE_MODE_MULTI_SEARCH);
}
```

In the `DEsearch` function of any class that is marked with `DE_MODE_MULTI_SEARCH`, you must NOT depend on the RDN value to determine whether it is a base search or one-level search. Instead, you need to check the scope from the `DMO_LDAP_Request`, and you will need to get the DN from the `DMO_LDAP_Request`.

```
SDKAccount::DEsearch(DMO_LDAP_Request *LDAP_request, ...)
{
    int scope = LDAP_request->getSearchScope();
    UTF8* searchBase = LDAP_request->getSearchBase();
    DistinguishedName searchDN(searchBase);
    UTF8* rdnValue = searchDN.getRDNvalue();
    If (scope == LDAP_SCOPE_BASE) {
        // base search
        // create the LDAP entry;
        *ppEntry = new DMO_LDAP_Entry(this, rdnValue);
    } else {
        // one-level search
        // same as before. No changes
        // When the search scope is one-level, there should not be any changes.
        // That is, don't use the search base from the search request. But
        continue
        // to use whatever you were using before as the search base.
    }
}
```

In `DEsearch`, when you create the `DMO_LDAP_Entry`, you must specify the `rdnValue`, shown above.

Build an Agent Plug-In

Perform the following procedure to build an agent plug-in.

To build your agent plug-in

1. Point your INCLUDE path to the directory that contains the OpenLDAP include files.
2. Point your INCLUDE path to the directory that contains the Provisioning SDK include files.

Link to the following required libraries:

- SuperagentBack.lib
- libslapd.lib
- libldap_r.lib
- liblutil.lib

Sample Call to Initialize the Agent Plug-In

The following is a sample of implementing the required `init_agent( )` function:

```
extern "C" {

DERepository EntryPointTable[] = {
    { NULL, "sampleNamespace", sampleNamespace_DEFactory },
    { NULL, "sampleDirectory", sampleDirectory_DEFactory },
    { NULL, "sampleAccountContainer", sampleAccountFolder_DEFactory },
    { NULL, "sampleAccount", sampleAccount_DEFactory },
    { NULL, "sampleGroupContainer", sampleGroupFolder_DEFactory },
    { NULL, "sampleGroup", sampleGroup_DEFactory },
    { NULL, NULL, NULL }
};

DModDirectoryEntry *
sampleNamespace_DEFactory(
    UTF8          *pszuObjectClass,
    UTF8          *pszuRdnType,
    UTF8          *pszuRdnValue
)
{
    return new sampleNamespace(pszuObjectClass, pszuRdnType, pszuRdnValue);
}
```

```
DMODirectoryEntry *
sampleDirectory_DEFactory(
    UTF8          *pszuObjectClass,
    UTF8          *pszuRdnType,
    UTF8          *pszuRdnValue
)
{
    return new sampleDirectory(pszuObjectClass, pszuRdnType, pszuRdnValue);
}

DMODirectoryEntry *
sampleAccountFolder_DEFactory(
    UTF8          *pszuObjectClass,
    UTF8          *pszuRdnType,
    UTF8          *pszuRdnValue
)
{
    return new sampleAccountFolder(pszuObjectClass, pszuRdnType,
pszuRdnValue);
}

DMODirectoryEntry *
sampleAccount_DEFactory(
    UTF8          *pszuObjectClass,
    UTF8          *pszuRdnType,
    UTF8          *pszuRdnValue
)
{
    return new sampleAccount(pszuObjectClass, pszuRdnType, pszuRdnValue);
}

DMODirectoryEntry *
sampleGroupFolder_DEFactory(
    UTF8          *pszuObjectClass,
    UTF8          *pszuRdnType,
    UTF8          *pszuRdnValue
)
{
    return new sampleGroupFolder(pszuObjectClass, pszuRdnType, pszuRdnValue);
}
```

```

DMODirectoryEntry *
sampleGroup_DEFactory(
    UTF8          *pszuObjectClass,
    UTF8          *pszuRdnType,
    UTF8          *pszuRdnValue
)
{
    return new sampleGroup(pszuObjectClass, pszuRdnType, pszuRdnValue);
}

/*=====+
    inti_agent:
        Return a pointer to the table of factory functions by setting
        the ppRep parameter.

        Return 0 for success.
        Return -1 for failure.
        Return DMO_AGENT_INVALID_LICENSE if license check fail.

    Remarks:
        Do not return LDAP error code.

+=====*/
int __declspec(dllexport)
init_agent(DERepository **ppRep)
{
    int          rc          =      0;

    if (ppRep == NULL) {
        rc = -1;
        goto exit;
    }
    *ppRep = EntryPointTable;

    /*
    || Do license check here. If the license check fail, return
    || DMO_AGENT_INVALID_LICENSE.
    */
    /*
    || Do any other initialization here.
    */

exit:
    return rc;
}

```

```
/*=====+
    shutdown_agent:
        Return 0 for success.
        Return -1 for failure.

    Remarks:
        Do not return LDAP error code.

+=====*/
int __declspec(dllexport)
    shutdown_agent()
{
    // Do any shutdown and/or clean up here.
    return 0;
}

} // extern "C"
```

Test the Agent Plug-in DLL

To test your plug-in using Manager, you must set the default= field of the AgentPluginDLL section of your parser table to the name of your agent plug-in DLL.

```
KEYWORD AgentPluginDLL
    attribute=eTAgentPluginDLL
    datalocation=agent
    edittype=string
    description=DLL for the Agent plug-in
    default=your_namespace_agent_plug-in_dll
    asciionly=yes
    verbreq=!add
    verbreq=!tomodify
```

your_namespace_agent_plug-in_dll

The DLL that you created for your agent plug-in.

Chapter 11: The Sample ODBC-SDK Connector

The SDK provides a sample ODBC-SDK connector that you can use as a template to develop new connectors for ODBC-accessible data. This chapter shows the steps for how to build and customize the sample.

The ODBC-SDK is built on top of the CORE-SDK. The ODBC-SDK is a pre-developed version of the CORE-SDK suitable for managing simple ODBC-accessible data.

The ODBC-SDK has a number of configuration options that make it an improvement over the CORE-SDK, including:

- A mechanism to minimize typing errors when assigning a TLA to a new connector
- Text strings are grouped in a shared header file, which makes it practical for developers to use the ODB.

Note: ODBC connectors provide compatibility with previous versions of Identity Manager. Where possible, we recommend using JDBC connectors instead. You can manage users in database tables using a JDBC connector. See the ConnectorXpress documentation for details.

More information:

[Provisioning SDK Guide Overview](#) (see page 15)

[Before you Begin Building the Sample Connector](#) (see page 55)

This section contains the following topics:

[How to Customize the Sample ODBC-SDK Connector](#) (see page 175)

[Create the KMS Connector](#) (see page 198)

[OdbcWrapper Class](#) (see page 200)

[How to Build the Sample ODBC-SDK Connector](#) (see page 203)

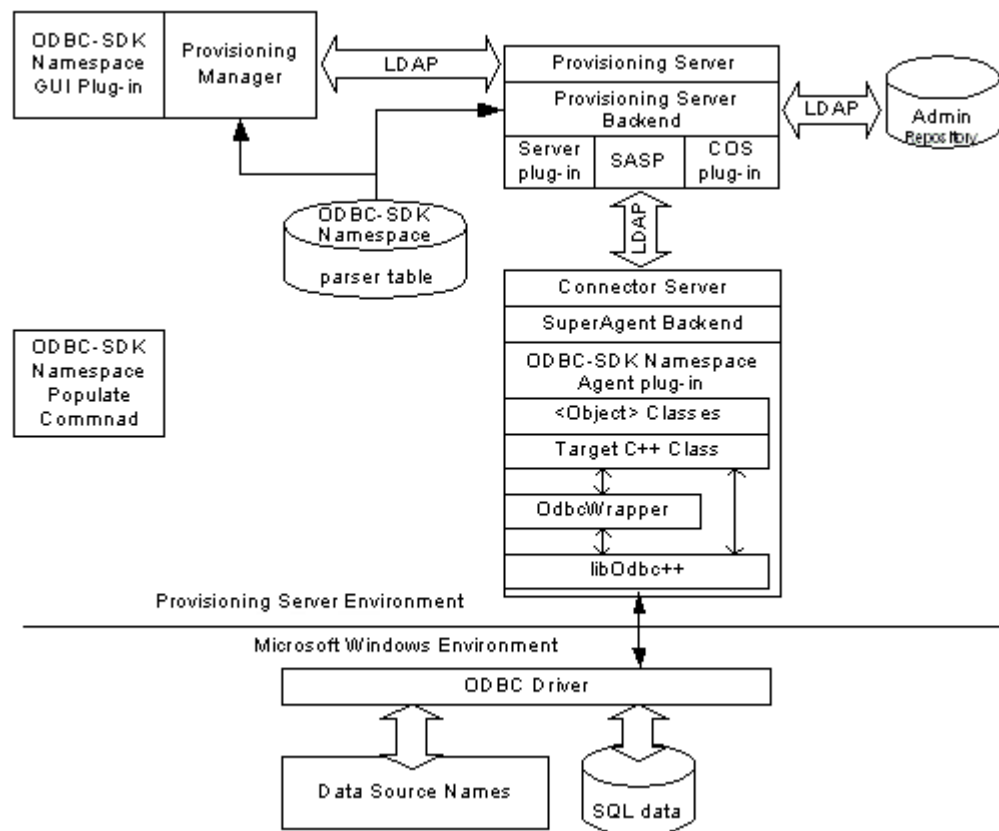
How to Customize the Sample ODBC-SDK Connector

The agent plug-ins can be customized to communicate with most ODBC sources. For simple database applications, only minor changes are required to complete a new ODBC-SDK Agent.

Customizing and implementing an ODBC-SDK connector in the Provisioning Server involves the following steps:

1. Analyze the database application.
2. Define the new schema objects.
3. Build and implement the GUI plug-in.
4. Customize and build the agent plug-in.
5. Update the populate command.
6. Configure SASP.

The following diagram illustrates the process:



Analyze the Database Application

The following tasks must be completed before customizing the ODBC-SDK connector:

- Identify the screen elements to be presented by the GUI.
- Plan the database schema.
- Map the screen elements to the database schema

The developer must be familiar with the database application that is being mapped to the Provisioning Server. It might be necessary to meet with the database administrator and the application owner to get the requested information. Having access to the application and the database itself can be useful as well.

Identify Screen Elements

Identify the screen elements to be presented by the application's Graphical User Interface (GUI). For each element, note the following:

- Field name
- Field data type
- Syntax and semantic, such as list of legal values, format, and so on
- GUI-control
- Form layout
- The relationship between pages if there are multiple pages
- Whether or not the GUI field is mandatory

This information should be available by examining the existing database application.

Identify Screen Elements - Example

This example describes the customization of the ODBC-SDK connector for a CA internal database application called KMS.

The following illustration shows the KMS web user interface:

The screenshot displays the KMS web user interface. On the left is a dark blue vertical sidebar with the 'ca' logo at the top, followed by 'Computer Associates', and a list of roles: 'Supervisors', 'Operators', 'View Only', and 'Graphic Admins'. Below this is a large 'KMS' logo. The main content area is titled 'Add User'. It features a 'Group:' dropdown menu with 'View Only' selected, and text input fields for 'First Name:', 'Last Name:', 'Login:', 'Password:', and 'Email:'. A 'Create' button is positioned below the 'Email' field. At the bottom of the form, there are two links: 'Back to User Administration' and 'Back To Main Menu'. A faint world map is visible in the background of the main content area.

The fields in the previous illustration are described in the following table:

GUI Field	Field Type	Notes
Group:	Pull-down selection list	Valid values: ■ supervisors ■ operators ■ view_only ■ graphics
First Name:	Text edit	
Last Name:	Text edit	
Login:	Text edit	Mandatory
Password:	Text edit	
Email:	Text edit	

Plan Database Schema

List the following information for the database schema for each table:

- Column name
- Data type
- Is it a primary key?
- Is it an indexed column?
- Is it a NOT NULL column?
- If multiple tables, identify the relationship between tables (if any)

You might need to get this information from the Database Administrator (DBA) who is responsible for maintaining the database application.

Note: The order of the columns in a table is important. Use an SQL query (SELECT ALL * FROM <tableName>) to get the column order (which is determined from left to right), or find the SQL CREATE TABLE statement used for creating it.

Plan Database Schema - Example

For the KMS application, account information is stored in a single table named PROFILES. The table is not indexed nor does it have a primary key but the account key in the table is the login column.

Column Name	Data Type	Notes
first_name	char(25)	
last_name	char(25)	
group_name	char(25)	
login	char(24)	Account key
password	char(24)	
email_id	char(75)	
comments	char(500)	

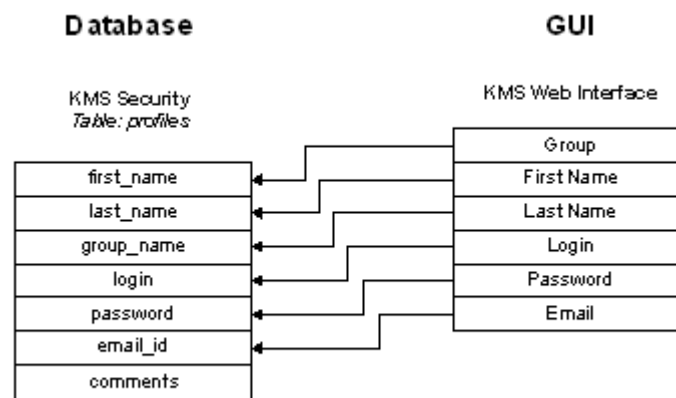
Map Screen Elements to Database Schema

Map each screen element to the corresponding element in the database schema. The mapping table should contain the GUI field name and the table and column name associated with each field name. As a minimum, the entries in this table should be the ones that can be manipulated with the new connector.

Note: All GUI field names must have a corresponding database field, and all database columns that are NOT NULL must have a corresponding GUI field. In both cases, an entry in the mapping table is necessary along with a note indicating the location of the data. This mapping table is needed for the integration code that is added in the development phase.

Map Screen Elements to Database Schema - Example

For the KMS application, the mapping is as follows:



GUI Field	Column Name	Data Type	Note
First Name:	first_name	char(25)	
Last Name:	last_name	char(25)	
Group:	group_name	char(25)	
Login:	login	char(24)	Account key
Password:	password	char(24)	
Email:	email_id	char(75)	
	comments	char(500)	Not mapped

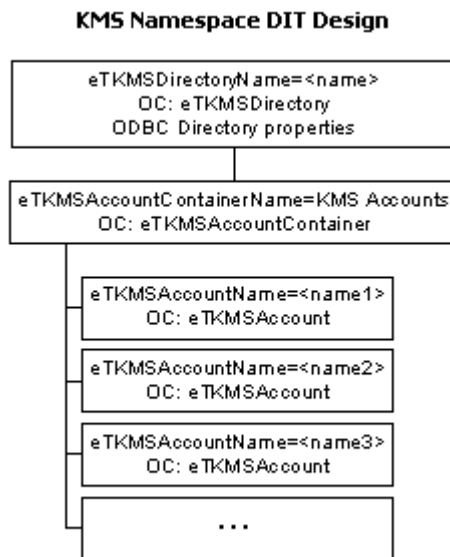
At this point you have enough information about the KMS application to define the schema.

Schema Objects Definition

To update the schema you must have a clear understanding of the objects in your endpoint type, their relationships and how to identify them. You must be familiar with the Directory Information Table (DIT), the parser table, and the relationships of endpoint objects to other objects, endpoints and endpoint types.

DIT Structure

The KMS endpoint type has the following structure:



Starting from the bottom of the endpoint type, the following objects appear:

- KMS Account object-Account objects are the Provisioning Server reflection of each account row in the KMS database. All account leaf objects belong to an object class of eTKMSAccount. An attribute must be identified or created that names this object in the KMS endpoint type.
- KMS Accounts container object-Account containers contain all the account objects that exist in a given endpoint. The KMS Accounts container object is a member of the eTKMSAccountContainer object class.

- KMS Directory container object-Directory containers contain all the objects of a endpoint. In the example above, the KMS Directory container object only contains a KMS Accounts container object. The KMS Directory container object is a member of the eTKMSDirectory object class.
- KMS Namespace container object-Namespace containers are top-level containers. They contain all the objects in the endpoint type. The KMS Namespace container object is a member of the eTNamespace object class.

Account Object

The KMS Account object is defined by associating pRovisiOning server attributes to each GUI field.

GUI Field	Column Name	Data Type	ETA2 Attribute	Note
First Name:	first_name	char(25)	eTKMSFirstName	
Last Name:	last_name	char(25)	eTKMSLastName	
Group:	group_name	char(25)	eTKMSGGroupName	
Login:	login	char(24)	eTKMSAccountName eTKMSLogin	Naming Attribute
Password:	password	char(24)	eTPassword	COS attribute
Email:	email_id	char(75)	eTKMSEmailId	
	comments	char(500)	eTKMSComments	Not mapped

Note: The login field is mapped to two attributes because it is the key in the table; it is mapped to the naming attribute of the account object as well as to its corresponding Provisioning Server attribute.

Global User Mappings

If the endpoint is used to populate the global users through the explore/correlate process, the appropriate mappings must be defined. The KMS Account object attributes map to the global user as follows:

Column Name	Namespace Attribute	Global Attribute	Note
first_name	eTKMSFirstName	eTFirstName	
last_name	eTKMSLastName	eTLastName	

Column Name	Namespace Attribute	Global Attribute	Note
group_name	eTKMSGGroupName	eTCustomField	
Login	eTKMSAccountName eTKMSLogin	eTGlobalUserName eTUserid	Account key
Password	eTPassword		COS attribute
email_id	eTKMSEmailId	eTEmailAddress	
Comments	eTKMSComments	eTDescription	Not mapped
	eTKMSFullName	eTFullName	Virtual Attribute

Note: A virtual attribute, eTKMSFullName, was added to the account definition and mapped to eTFullName. This is required for the Provisioning Server correlation engine to work efficiently. If the endpoint does not need to be correlated, the virtual attribute may not be necessary.

Global User to Connector Mappings

This mapping defines the default policy that is created when the endpoint is initialized. The default KMS policy maps global user attributes to the KMS account object attributes as follows:

Global Attribute	Rule String	Namespace Attribute	Note
eTFirstName	%UF%	eTKMSFirstName	
eTLastName	%UL%	eTKMSLastName	
eTCustomField01	%UCU01%	eTKMSGGroupName	
eTGlobalUserName	%AC%	eTKMSAccountName eTKMSLogin	Account key
eTPassword	%P%	eTPassword	COS attribute
eTEmailAddress	%UE%	eTKMSEmailId	
eTDescription	%UD%	eTKMSComments	Not mapped
eTFullName		eTKMSFullName	Virtual Attribute

Parser Table

The parser table is the main source of schema information. It defines the object classes and associated attributes used for presenting the ODBC-SDK connector-specific information. The following is an example of the KMS account parser file (SDKAccount_Property.pti):

```
#
# ODBC-SDK: Sample KMS PROFILE tables account properties
#

KEYWORD first_name
  minabbrev=10
  attribute=eTKMSFirstName
  datalocation=Agent
  edittype=string
  editflag=yes
  minlen=
  maxlen=25
  minvalue=
  maxvalue=
  asciionly=yes
# JIAM updates
#   Obscured=no
#   Deprecated=no
#   description=KMS first name
#   default=
#   Hidden=no
#   Verbreq=

KEYWORD last_name
  minabbrev=9
  attribute=eTKMSLastName
  datalocation=Agent
  edittype=string
  editflag=yes
  minlen=
  maxlen=25
  minvalue=
  maxvalue=
  asciionly=yes
# JIAM updates
#   Obscured=no
#   Deprecated=no
#   description=KMS last_name
#   default=
#   hidden=no
#   Verbreq=
```



```
KEYWORD group_name
    minabbrev=10
    attribute=eTKMSGGroupName
    globaluserattribute=eTCustom
    datalocation=Agent
    edittype=string
    editflag=yes
    minlen=
    maxlen=25
    minvalue=
    maxvalue=
    asciionly=yes
# JIAM updates
# Obscured=no
# Deprecated=no
    description=KMS group_name
    default=
    hidden=no
# Verbreq=

KEYWORD login
    minabbrev=5
    attribute=eTKMSLogin
        globaluserattribute=eTGlobalUserName
        globaluserattribute=eTUserId
    datalocation=Agent
    edittype=string
    editflag=yes
    minlen=
    maxlen=24
    minvalue=
    maxvalue=
    asciionly=yes
# JIAM updates
# Obscured=no
# Deprecated=no
    description=KMS login
    default=
    hidden=no
    Verbreq=ADD

KEYWORD email_id
    minabbrev=8
    attribute=eTKMSEmailId
    globaluserattribute=eTEmailAddress
    datalocation=Agent
    edittype=string
    editflag=yes
    minlen=
```

```
        maxlen=75
        minvalue=
        maxvalue=
        asciionly=yes
# JIAM updates
#   Obscured=no
#   Deprecated=no
        description=KMS_email_id
        default=
        hidden=no
#   Verbreq=

        KEYWORD_comments
        minabbrev=8
        attribute=eTKMSComments
        globaluserattribute=eTDescription
        datalocation=Agent
        edittype=string
        editflag=yes
        minlen=
        maxlen=500
        minvalue=
        maxvalue=
        asciionly=yes
# JIAM updates
#   Obscured=no
#   Deprecated=no
        description=KMS_comments
        default=
        hidden=no
#   Verbreq=

#
# Virtual attributes
#
        KEYWORD_full_name
        minabbrev=8
        attribute=eTKMSFullName
        globaluserattribute=eTFullName
        datalocation=Agent
        edittype=string
        editflag=yes
        minlen=
        maxlen=51
        minvalue=
        maxvalue=
        asciionly=yes
# JIAM updates
#   Obscured=no
```

```
#   Deprecated=no
      description=KMS full_name*
      default=
      hidden=no
#   Verbreq=
```

Accounts Container Object

The accounts container object remains essentially identical to the ODBC-SDK sample, except that the OSN acronym needs to be replaced by the appropriate new connector value. The following is an example of the accounts container extract of the KMS parser file (SDKParse.pty):

```
#####
#   ODBC-SDK Account Container Class
#
#   Note the following parameters must be updated:
#       CLASS
#       ditparentclass
#       nameproperty
#       win32dll
#       containerClass
#       extname
#####
CLASS KMSAccountContainer,eTKMSAccountContainer,etavlcors,secobjar
    smplugindll=saspServerPlugin
    ditparentclass=eTKMSDirectory
    hidden=no
    nameproperty=eTKMSAccountContainerName
    win32dll=KMSGUI
    win32icon=IDI_ODBC_ACCOUNT
    win32menu=IDR_SDKACCCONTAINER_MENU
    containerclass=eTKMSDirectory
    draggable=no
    expandable=yes
    sheet=no
    predefined=yes
    extname=KMS Accounts
```

```
KEYWORD Name
    minabbrev=4
    attribute=eTKMSAccountContainerName
    datalocation=both
    editttype=string
    editflag=yes
    minlen=1
    maxlen=255
    minvalue=
    maxvalue=
    hidden=no
    description=KMS Account Container name
    default=
    asciionly=yes
    verbreq=ADD
#
# Get the common object attributes
#
#include ".././../include/pti/comobject.pti"
```

Directory Container Object

The directory container object is essentially identical to the ODBC-SDK sample except that the OSN acronym needs to be replaced by the appropriate new connector value. The following shows the SDKDirectory_Property.pti file updated for the KMS Namespace:

```
#
# ODBC-SDK Directory Properties
#
KEYWORD odbcsdk_comments
    minabbrev=11
    attribute=eTKMSDirectoryComments
    datalocation=both
    edittype=string
    editflag=yes
    minlen=
    maxlen=128
    minvalue=
    maxvalue=
    asciionly=no
# JIAM updates
# Obscured=no
# Deprecated=no
    description=ODBC-SDK comments
    default=
    Hidden=no
# Verbreq=

KEYWORD dsn
    minabbrev=3
    attribute=eTKMSDirectoryDSN
    datalocation=both
    edittype=string
    editflag=yes
    minlen=1
    maxlen=100
    minvalue=
    maxvalue=
    asciionly=yes
#
```

```
# JIAM updates
#   Obscured=no
#   Deprecated=no
#   description=ODBC Data source name (DSN)
#   default=
#   Hidden=no
#   Verbreq=ADD

KEYWORD uid
  minabbrev=3
  attribute=eTKMSDirectoryUID
  datalocation=both
  edittype=string
  editflag=yes
  minlen=1
  maxlen=100
  minvalue=
  maxvalue=
  asciionly=yes
# JIAM updates
#   Obscured=no
#   Deprecated=no
#   description=ODBC UserID
#   default=
#   Hidden=no
#   Verbreq=

KEYWORD pwd
  minabbrev=3
  attribute=eTKMSDirectoryPWD
  datalocation=both
  edittype=string
  editflag=yes
  minlen=1
  maxlen=128
  minvalue=
  maxvalue=
  asciionly=no
  encrypted=yes          encryptwith=eTKMSDirectoryDSN,eTKMSDirectoryUID

# JIAM updates
#   Obscured=no
#   Deprecated=no
#   description=ODBC password
#   default=
#   Hidden=no
#   Verbreq=
```

Endpoint Container Object

The endpoint container object is almost identical to the ODBC-SDK sample. The OSN acronym must be replaced by the appropriate new endpoint value.

Build and Implement the GUI Plug-In

A GUI plug-in is a runtime loadable DLL that manages users, accounts, and other objects in a connector.

Follow the steps for building your GUI plug-in.

More information:

[Building a GUI Plug-In](#) (see page 163)

The following illustration shows the account property page used to convey KMS account data:



The screenshot shows a dialog box titled "KMS Account". It contains several input fields, each preceded by an asterisk (*):

- * Group : A dropdown menu.
- * First name : A text input field.
- * Last name : A text input field.
- * Login : A text input field.
- * Password : A text input field.
- * Email : A text input field.
- Comments : A larger text area with a vertical scrollbar on the right.

Build and Implement the Agent Plug-in

Making the required changes to the ODBC-SDK Connector agent code is as simple as the ODBC application.

More information:

[Building an Agent Plug-In](#) (see page 163)

Attribute Mapping Table

The OdbcWrapper class uses this table to automatically generate SQL queries and parse their results.

More information:

[OdbcWrapper Class](#) (see page 200)

The following code sample shows an attribute mapping table for the KMS PROFILES table:

```
// Attribute Mapping Tables
static
AttributeMap tAccountTableMap[] = {
/*
|| **** IMPORTANT: Order of the following records is very important!****
||           They must be in the same order as the columns in the target table
*/
//Attribute Name      Column Name  Data Type  MapToSQL?  MapToLDAP?  PrimKey?
{ UTFODBC_ACC_FIRSTNAME, "FIRST_NAME",SQL_CHAR, true,    true,    false },
{ UTFODBC_ACC_LASTNAME,  "LAST_NAME", SQL_CHAR, true,    true,    false },
{ UTFODBC_ACC_GROUP,     "GROUP_NAME",SQL_CHAR, true,    true,    false },
{ UTFODBC_ACC_LOGIN,     "LOGIN",    SQL_CHAR, true,    true,    true  },
{ UTFODBC_ACC_PASSWORD,  "PASSWORD", SQL_CHAR, true,    true,    false },
{ UTFODBC_ACC_EMAIL,     "EMAIL_ID", SQL_CHAR, true,    true,    false },
{ UTFODBC_ACC_COMMENTS,  "COMMENTS", SQL_CHAR, true,    true,    false },
/*
|| **** IMPORTANT: Virtual Attributes go here -- with "" as the column name
*/
#ifdef UTFODBC_ACC_FULLNAME
{ UTFODBC_ACC_FULLNAME, "",                -1, false,    true,    false },
#endif // UTFODBC_ACC_FULLNAME
/*
|| **** IMPORTANT: Next entry is not a column but the
||           table name along with the C index of
||           the Primary Key column
*/
{ NULL,                "PROFILES",        3, false,    false,    false },
/*
|| **** IMPORTANT: End of table Marker
*/
{ NULL }
};
```


Virtual Attributes

Additional code is required to create virtual attributes when they are requested by the Provisioning Server. The method that needs to be modified is `Target::SendDataToEta()` in `Agent/Target.cpp`.

Usually, virtual attributes have a value that is derived by manipulating one or more values from a search result.

Virtual attributes are often used to create a connector-specific attribute to convey the user's full name when such a field is not directly available from the target connector but can be constructed from two or more fields.

To implement virtual attributes, the following modifications must be applied to the `SendDataToEta()`:

- Save the values needed to build the virtual value from the fields as they appear when parsing the column data.
- When the column parsing has been completed, generate the value from the saved data and assign it to the virtual attribute.

Virtual Attributes - Example

The following shows the handling of a KMS eTKMSFullName virtual attribute:

```
intTarget::SendDataToEta(
    OdbcWrapper          &odbcIf,
    odbc::ResultSetMetaData *pMetaData,
    odbc::ResultSet      *pResultSet,
    string               sqlNamingColumn,
    string               suAccount,
    UTF8                 **ppszuAttrList,
    DMOSearchOp          *pSearchOp,
    DMODirectoryEntry    *pDE,
    string               &sErrMsg    /* out */
)
{
    int          iColumnIndex;
    int          iColumnCount;
    string       sAttrName;
    string       sColumnName;
    string       sValue;
#ifdef UTFODBC_ACC_FULLNAME
    string sFirstName = "";
    string sLastName  = "";
    string sFullName  = "";
#endif // UTFODBC_ACC_FULLNAME

    iColumnCount = pMetaData->getColumnCount(); // Number of columns in each row
```

```

/*
|| Cycle through each returned row data
*/
while ((pResultSet)->next()) {

    // Create a new ETA Ldap entry
    if (suAccount.empty()) {
        // Use the value received from the table.
        sValue = odbcIf.getLdapValueFromSqlColumn(pResultSet, sqlNamingColumn);
    } else {
        // Use the value provided by eTrust Admin.
        sValue = suAccount;
    }
    if (sValue.empty()) {
        continue;    // Note: Could be flagged as an error
    }
    DMO_LDAP_Entry oEntry(pDE, (UTF8*)sValue.c_str());

/*
|| Cycle through each column of this row...
*/
for (iColumnIndex = 1; iColumnIndex <= iColumnCount; iColumnIndex++) {

    // Current column name & value
    sColumnName = pMetaData->getColumnName(iColumnIndex);
    sValue = odbcIf.getLdapValueFromSqlColumn(pResultSet, sColumnName);

/*
|| Straight LDAP-to-SQL attribute/column map
*/
if (odbcIf.isSqlColumnMappedToLdap(sColumnName)) {
    if (sValue.length() == 0) {
        // skip if the value is null
        continue;
    }

    sAttrName = odbcIf.getLdapAttrFromSqlColumn(sColumnName);
    if (isInAttrList(sAttrName, ppszAttrList)) {
        oEntry.AddAttribute((UTF8*)sAttrName.c_str(), (UTF8*)sValue.c_str());
    }
    // Fall-through: in case we need to save some values for later
}

/*
|| Virtual Attributes: Save values for later
*/
#ifdef UTFODBC_ACC_FULLNAME
    if (strcmp(sColumnName.c_str(), "FIRST_NAME") == 0) {
        sFirstName = sValue;
    }
}

```

```
    }
    if (strcmp(sColumnName.c_str(), "LAST_NAME") == 0) {
        sLastName = sValue;
    }
#endif // UTFODBC_ACC_FULLNAME
} // end for loop

/*
|| Handle "virtual" attributes
*/
#ifdef UTFODBC_ACC_FULLNAME
if (isInAttrList(UTFODBC_ACC_FULLNAME, ppszAttrList)) {
    if (sFirstName.empty() && sLastName.empty()) {
        continue;
    }
    sFullName = rtrim(sFirstName.empty() ? sLastName : /
        sFirstName + " " + sLastName);
    pCurE->addAttribute(UTFODBC_ACC_FULLNAME, (UTF8*) sFullName.c_str());
}
#endif // UTFODBC_ACC_FULLNAME

/*
|| Send the entry.
*/
pSearchOp->SendEntry(&oEntry);
}
return LDAP_SUCCESS;
}
```

Update the Populate Command

The populate command needs to be updated for the schema changes and the new default policy required by the new endpoint.

The following shows the KMS default policy, an extract from Populate/SDKpop.h:

```
{ DN4defaultPolicy,
    {
        {"objectClass",          "eTKMSPolicy"},
        // Policy details
        {"eTKMSPolicyName",      "KMSDefaultPolicy"},
        {"eTDescription",        "KMS Default Policy"},
        {"eTNameSpaceType",      "eTKMSDirectory"},
        // Entry Template
        {"eTKMSAccountName",     "%AC%"},
        {"eTKMSGGroupName",      "View_only"},
        {"eTKMSFirstName",       "%UF%"},
        {"eTKMSLastName",        "%UL%"},
        {"eTKMSLogin",           "%AC%"},
        {"eTPassword",           "%P%"},
        {"eTKMSEmailId",         "%UE%"},
        {"eTKMSComments",        "%UD%"},
        // Other policy details
        {"eTAccountStatus",      "A"},           // active
        {"eTPropagateChanges",    "0"},
        {"eTStrongSync",          "0"},
        {"eTSuspended",           "0"},
        NULL
    } /* policy */
},
```

Configure the Superagent Server Plug-In

More information:

[Implementing Connectors](#) (see page 48)

Configure the Superagent Server Plug-In - Example 1

The following code sample shows the KMS server plug-in definition for SASP:

```
#
# Namespace for KMS objects
#
NAMESPACE KMS Namespace
    smplugindll=saspRegister
```

Configure the Superagent Server Plug-In - Example 2

The following code sample shows the KMS agent plug-in DLL name defined in the Namespace Class definition in the parser table:

```
KEYWORD AgentPluginDLL
  attribute=eTAgentPluginDLL
  datalocation=agent
  edittype=string
  description=DLL for the ODBC-SDK Agent plug-in
  default=KMSNamespace.dll
  asciionly=yes
  verbreq=!ADD
  verbreq=!TOUPDATE
#
# Announce that the ODBC-SDK namespace (connector)
# validates its directory credentials in its
# agent plug-in's modify function and doesn't need this
# behavior to be controlled by server configuration parameter
# "Validate Directory Credentials".
#
#include "../include/pti/NS_ValidateCredentials.pti"
```

Create the KMS Connector

Perform the following procedure to create the KMS connector.

To create the KMS connector

1. Perform the database application analysis.
2. Copy the *PSDKHOME/Samples/ODBC-SDK* and its subfolders to *PSDKHOME/Samples/KMS*.
3. Edit *PSDKHOME/Samples/KMS/Namespace.inc*, change the MYNAME value from OSN to KMS.

4. Open the GUI Project and update the parser table to reflect the KMS endpoint type:
 - Edit SDKparser.pti and change all attribute names from eTOSN to eTKMS. Change OSN references to KMS (such as win32dll).
 - Update SDKAccount_Property.pti, remove unneeded attributes and add KMS specific attributes. Use the information gathered from the analysis:
 - Attribute length
 - Global user mapping for the explore/correlate process
 - Mandatory attributes
 - Virtual attributes
 - Value lists
 - Update SDKDirectory_Property.pti by changing eTOSN to eTKMS.
5. Update SDKschema.h by replacing the value of the UTFPI macro from OSN to KMS, and adding macros for all the attributes defined in the SDKAccount_Property.pti. Remove unneeded macros. Examine the macros that end with _VALUE or _DEFVALUE, and update their values with the appropriate ones for the KMS connector.
6. Update the SDKclean.bat file in the KMS connector.
7. Create a new GUI property page along with the supporting code. Used the macros defined in SDKschema.h when referring to Provisioning Server attributes.
 - Ensure that the class is a subclass of CPropertyPage and that the DECLARE_DYNAMIC() / IMPLEMENT_DYNAMIC() macros are called in the class header and class implementation files. Use AccGeneralPage.cpp and AccGeneralPage.h as templates.
 - Ensure that the setData(), apply() and setErrorField() methods are implemented.
 - Update the resource string table for IDS_DIRECTORIES and IDS_POLICIES to reflect the KMS connector.

8. In the OdbcNamespace project, modify Target.cpp with a new AttributeMap array reflecting the KMS application. Add extra code to handle any virtual attributes.
9. In the Populate project, update the Populate/SDKpop.h for the KMS connector and create a default policy for KMS. Remove the default policy for OSN.
10. Run nmake in the top-level folder of the new connector to create the following files in the *PSDKHOME/bin* and *PSDKHOME/data* directories:
 - KMSGui.dll
 - KMSparse.ptt
 - KMSNamespace.dll
 - KMSpop.exe
 - KMSclean.bat

OdbcWrapper Class

Each database table is represented by an array of AttributeMap objects. The OdbcWrapper class and its associated AttributeMapIF class provide a table-driven interface to the libOdbc++ library. Each AttributeMap object represents a database column and its properties.

Note: The order of the objects in the array is critical. The objects must be in the order of the columns defined in the corresponding ODBC table. The last object of the array is special and it encodes the ODBC table name.

The OdbcWrapper class uses the information encoded in the array of AttributeMap objects to search, modify and delete information from the target ODBC database.

More information:

[OdbcWrapper Class Method](#) (see page 301)

AttributeMap

The following example is the array representing the sample OSN connector, which manipulates the ODB_ACCESS table where A_EMAIL is the naming column:

```
// Attribute Mapping Tables
// ODBC-SDK Sample access table
AttributeMap tAccountTableMap[] = {/*
|| **** IMPORTANT: Order of the following records is very important!****
||          They must be in the same order as the columns in the target table
*/
// Attribute Name      Column Name  Data Type  MapToSQL? MapToLDAP?  PrimKey?
{ UTFODBC_ACC_ID,      "A_ID",      SQL_CHAR,  false,    false,    false },
{ UTFODBC_ACC_ACCOUNTID,"A_ACCOUNT_ID",SQL_CHAR,false,    false,    false },
{ UTFODBC_ACC_GROUP,   "A_GROUP",   SQL_CHAR,  true,     true,     false },
{ UTFODBC_ACC_PASSWORD,"A_PASSWORD",SQL_CHAR,  true,     true,     false },
{ UTFODBC_ACC_EMAIL,   "A_EMAIL",   SQL_CHAR,  true,     true,     true  },
{ UTFODBC_ACC_COMPANY, "A_COMPANY", SQL_CHAR,  true,     true,     false },
{ UTFODBC_ACC_CONTACT, "A_CONTACT", SQL_CHAR,  true,     true,     false },
{ UTFODBC_ACC_AREACODE, "A_AREACODE",SQL_CHAR,  true,     true,     false },
{ UTFODBC_ACC_PHONE,   "A_PHONE",   SQL_CHAR,  true,     true,     false },
{ UTFODBC_ACC_EXTN,    "A_EXTN",    SQL_CHAR,  true,     true,     false },
{ UTFODBC_ACC_INTERPHONE,"A_INTERPHONE",SQL_CHAR,true,     true,     false },
{ UTFODBC_ACC_FAX,     "A_FAX",     SQL_CHAR,  true,     true,     false },
{ UTFODBC_ACC_CERTPRINTER,"A_CERTPRINTER_VER",SQL_INTEGER,false,false,false},
/*
|| **** IMPORTANT: Virtual Attributes go here -- with "" as the column name
*/
#ifdef HANDLE_ETXSUSPENDED
{ ETXSUSPENDED,        "",                -1,  false,  false,  false },
#endif // eTxsuspended
/*
|| **** IMPORTANT: Next entry is not a column but the
||          table name along with the C index of
||          the Primary Key column
*/
{ NULL,                "ODB_ACCESS",      4,  false,  false,  false },
/*
|| **** IMPORTANT: End of table Marker
*/
{ NULL }
}
```

How to Install LibODBC++ and Configure the OSN Connector

The ODBC-SDK connector provided by the Provisioning SDK relies on the Open License package libodbc++. If you want to use the ODBC-SDK connector feature, you must download the ODBC package. Otherwise, you will not have the Libodbc++ library required to build the sample ODBC-SDK sample connector. The ODBC package is a C++ class library for accessing SQL databases. It provides a subset of the JDBC 2.0 (tm) and runs on top of ODBC.

Note: libodbc++ is no longer bundled with the SDK.

1. Before you implement the ODBC-SDK Connector, you must download the LibOdbc++ source from <http://libodbcxx.sourceforge.net/>.
2. Un-archive the LibOdbc++ to your hard drive.

```
LIBODBCXX_PATH=C:/Work/libodbc++-0.2.3
```

The LIBODBCXX_PATH environment variable is used throughout these steps to refer to the installation path of the LibOdbc++ source.

3. The build procedures are in the %LIBODBCXX_PATH%/INSTALL file. However, the Win32 makefile must first be updated with the appropriate build flags to build a LibOdbc++ library for use with the Provisioning SDK. The two most useful build configurations are debug-lib, which builds %LIBODBCXX_PATH%/win32/debug-lib/odbc++sd.lib, and prod-lib, which builds %LIBODBCXX_PATH%/win32/prod-lib/odbc++s.lib.

Depending upon the configuration that you choose, make one of the following changes:

- 3.1 CFG=debug-lib

Replace:

```
DEBUG_CXXFLAGS=/Zi /DODBCXX_DEBUG
```

with

```
DEBUG_CXXFLAGS=/Zi /DODBCXX_DEBUG -Od -c -W3 -D_X86_=1 -  
D_WINNT -DWIN32 -D_WIN32 -D_MT -D_DLL -MD /GX /GR /D"MBCS"  
/D"_MBCS"
```

- 3.2 CFG=prod-lib

Replace:

```
PROD_CXXFLAGS=/O2
```

with

```
PROD_CXXFLAGS=/O2 -c -W3 -D_X86_=1 -D_WINNT -DWIN32 -  
D_WIN32 -D_MT -D_DLL -MD /GX /GR /D"MBCS" /D"_MBCS"
```

4. Build the library, as documented in the %LIBODBCXX_PATH%/INSTALL file. For example:

```
C:> cd %LIBODBCXX%  
C:> cd win32  
C:> nmake /f makefile.w32 CFG=debug-lib
```

5. Note the path to the library, as you will need it to update the ODB configuration files. For example:

```
LIBODBCXX_LIB=C:/Work/libodbc++-0.2.3/win32/debug-lib/odbc++sd.lib
```

6. Configure the ODB Connector for the library to use by updating the LIBODBCXX_* variables in PSDKHOME/Samples/ODBC-SDK/Namespace.inc. The following is an example:

```
#-----  
# Local LibOdbc++ library  
#-----  
# Path to the LibOdbc++ installation folder  
LIBODBCXX_PATH=C:/Work/libodbc++-0.2.3  
#  
# Path to the LibOdbc++ header files  
LIBODBCXX_INC=$(LIBODBCXX_PATH)/include  
#  
# Full path to the LibOdbc++ library  
LIBODBCXX_LIB=$(LIBODBCXX_PATH)/win32/debug-lib/odbc++sd.lib
```

How to Build the Sample ODBC-SDK Connector

The major steps for building and integrating the sample ODBC-SDK connector are listed below:

1. Download and build the libOdbc++ library.
2. Compile the sample ODBC-SDK connector.
3. Add the ODBC connector object to the Provisioning Directory.
4. (Optional) Create a sample ODBC-SDK database.
5. (Optional) Create a new ODBC-SDK Data Source Name (DSN).
6. Register the sample directory in the ODBC-SDK connector.
7. Explore and correlate the accounts on the ODBC-SDK Directory.

Note: OSN is the acronym for the ODBC-SDK connector.

Download and Build the libOdbc++ Library

The ODBC-SDK connector relies on the Open License package libOdbc++. If you want to use the ODBC-SDK connector feature, you must download the ODBC package. Otherwise, you will not have the LibOdbc++ library required to build the sample ODBC-SDK sample connector. The ODBC package is a C++ class library for accessing SQL databases. It provides a subset of the JDBC 2.0 (tm) and runs on top of ODBC.

Note: The libOdbc++ is no longer bundled with the SDK.

To download and build the library

1. Download the libOdbc++ source from <http://libodbcxx.sourceforge.net/>
2. Un-archive the libOdbc++ on your drive. Note the location. The following is an example:

```
LIBODBCXX_PATH=C:\Work\libodbc++-0.2.3
```

Note: The LIBODBCXX_PATH environment variable is used in these procedures to refer to the installation path of the LibOdbc++ library source.

The detailed build procedures are in the %LIBODBCXX_PATH%/INSTALL file. However, you must first update the Win32 makefile with the appropriate build flags to build a LibOdbc++ library compatible for use with the Provisioning SDK. The two build configurations that are most useful are "debug-lib" and "prod-lib". They build %LIBODBCXX_PATH%/win32/debug-lib/odbc++sd.lib and %LIBODBCXX_PATH%/win32/prod-lib/odbc++s.lib.

3. Change the build flags in the %LIBODBCXX_PATH%/win32/makefile.w32.

For the debugging version of the library (Target CFG=debug-lib), replace the following:

```
DEBUG_CXXFLAGS=/Zi /DODBCXX_DEBUG
```

with

```
DEBUG_CXXFLAGS=/Zi /DODBCXX_DEBUG -Od -c -W3 -D_X86_=1 -D_WINNT -DWIN32 -  
D_WIN32 -D_MT -D_DLL -MD /GX /GR /D"MBCS" /D"_MBCS"
```

For the production version of the library (Target CFG=prod-lib), replace the following:

```
PROD_CXXFLAGS=/O2
```

with

```
PROD_CXXFLAGS=/O2 -c -W3 -D_X86_=1 -D_WINNT -DWIN32 -D_WIN32 -D_MT -D_DLL -MD  
/GX /GR /D"MBCS" /D"_MBCS"
```

4. Build the library, as documented in the %LIBODBCXX_PATH%/INSTALL file, as shown in the following example:

```
C:> cd %LIBODBCXX_PATH%
C:> cd win32
C:> nmake /f makefile.w32 CFG=debug-lib
```

5. Note the path to the library, as you need it to update the ODB configuration files, for example:

```
LIBODBCXX_LIB=C:\Work\libodbc++-0.2.3\win32\debug-lib\odbc++sd.lib
```

6. Configure the OSN Connector for the library to use by updating the LIBODBCXX_* variables in PSDKHOME/ Samples/ODBC-SDK/Namespace.inc. For example:

```
#-----
# Local LibOdbc++ library
#-----
# Path to the LibOdbc++ installation folder
LIBODBCXX_PATH=C:\Work\libodbc++-0.2.3
#
# Path to the LibOdbc++ header files
LIBODBCXX_INC=$(LIBODBCXX_PATH)/include
#
# Full path to the LibOdbc++ library
LIBODBCXX_LIB=$(LIBODBCXX_PATH)/win32/debug-lib/odbc++sd.lib
```

Compile the Sample ODBC-SDK Connector

Perform the following procedure to compile the sample ODBC-SDK connector.

To compile the sample connector

1. Open a command prompt and change to the following directory:

```
PSDKHOME/Samples/ODBC-SDK
```

2. Run NMAKE.

Note: To verify that your development environment settings are correct, run the VSvars32.BAT file before you run NMAKE.

The following files are generated. You can find these files in the PSDKHOME/bin and PSDKHOME/data directories:

- OSNGUI.dll
- OSNParse.ptt
- OSNNamespace.dll
- OSNpop.exe
- OSNclean.bat

Add the ODBC-SDK Connector Object to Manager

Perform the following procedure to add the ODBC-SDK Connector object in Provisioning Manager.

To add the ODBC-SDK Connector object in Provisioning Manager

1. Double-click the Services icon in the Control Panel. The Services window appears.
2. Stop the Provisioning Server and the superagent.
3. Click Close.
4. Close and exit from your current Provisioning Manager session.
5. Copy the following files from *PSDKHOME*/bin to *PSHOME*/bin:
 - OSNGUI.dll
 - OSNpop.exe
 - OSNNamespace.dll
 - OSNclean.bat
6. Copy the OSNParse.ptt file from *PSDKHOME*/data to *PSHOME*/data.
7. Generate the schema for the new connector, as follows:
 - Create the Provisioning Server schema by running:

```
C:> schemagen -genoid -n OSN:15000
```

This creates the following files in *PSHOME*/data:

```
etrust_osn.dxc  
etrust_osn.schema
```
 - Update the Provisioning Server schema.

From the Services panel, stop Provisioning service (should already be stopped), eTrust Directory etrustAdmin, and eTrust Directory etadmintemp.

Add the following line to *PSHOME*/data/etrust_admin.conf:

```
include "C:\\Program Files\\CA\\Provisioning  
Server\\Data\\etrust_osn.schema"
```
 - Update the eTrust Directory schema.

Copy etrust_osn.dxc to %DXHOME%/config/schema

Add the following line to
%DXHOME%/config/schema/etrustadmin.dxc:

```
source "etrust_osn.dxc";
```

- Update the %DXHOME%/Config/Database/eTrustAdmin.dxc file, adding to the parameter "set index-wide =" the name of the eTOSNDirectoryDSNlist attribute, separating with from other attribute names by a comma.

Save the eTrustAdmin.dxc file.

- Update the PSHOME/bin/etaindex.bat file, adding the name of the eTOSNDirectoryDSNlist attribute at the end of the line "xindexdb etrustadmin +wide".

Save the file, and restart the eTrustAdmin dxserver service.

Run the etaindex.bat file.

Note: Running the etaindex.bat file is not required when the eTOSNDirectoryDSNlist attribute is new or has no stored value.

More information:

[Implementing Connectors](#) (see page 48)

8. Restart the Provisioning service and the superagent.
9. Open a command prompt and change to the following directory:

PSHOME/bin

10. Enter the following command:

```
OSNpop <domain_name> <administrator_name> <administrator_password>
```

11. Start the Provisioning Manager.

Create a Sample ODBC Database (Optional)

You can optionally create a sample ODBC database.

To create a sample Advantage™ Ingres® database named ODBC-SDK that contains the tables the sample OSN connector can explore and manipulate, do the following:

```
C:> cd "PSDKHOME/Samples/ODBC-SDK/sampleData"  
C:> MakeDatabase.bat
```

Note: You can also use makefile, by running "nmake load."

Working with Ingres R3 or above

You can check the version of Ingres that runs on your machine by opening the `version.rel` file, located in the folder that corresponds to the environment variable `II_SYSTEM`, and check whether the line contains `II 2` or `II 3`.

If your installation is using Ingres R3 or above, the sample database must be created with an Ingres user account with the relevant rights. If you do not have the proper rights to execute this step, ask the administrator of your machine to do it for you.

Note: On Windows, the administrator as well as the user who installed the Provisioning Server (they may be the same) are Ingres users by default.

Create a New ODBC Data Source Name (Optional)

If you create a sample database or if you have an existing database that does not have a DSN, you must create a DSN for that database. The following steps are to create a DSN for an Advantage Ingres database. Steps to create a DSN for another RDBMS are similar but not identical.

To create a DSN for a database

1. Go to the start panel, Settings and choose Control Panel. Depending on the host operating system, you will find the Data Source/ODBC Applet in this folder or in the Administrative Tools sub-folder.
2. Once the ODBC Data Source Administrator is up, select the System DSN tab, and click the Add button.
3. Select the driver you need to connect to the target database. In this case, you want the Ingres driver.
4. The Ingres ODBC Administrator panel opens. Enter a name for your data source, select (LOCAL) as the Vnode, and leave INGRES as the Type.
5. The database name is the actual database you want to connect to (in this case ODBC-SDK). Click Apply.
6. Optionally, you can click the Test button to verify that everything is okay.
7. Click OK to save the new Data Source Name (DSN).

Register the Sample ODBC-SDK Endpoint

Perform the following procedure to register the sample endpoint.

To register the sample ODBC-SDK endpoint

1. Select the Namespaces task frame.
2. Select the OSN Directory object type.
3. Click New.
4. Enter the following values in the ODBC-SDK Directory dialog:
 - *Directory Name—ODBC Directory
 - *Data Source Name—Name of the ODBC database
 - Comments—"ODBC-SDK Namespace Test Database"

If your installation is using Ingres R3 or above, you must also supply the following authentication fields:

- User name—Bind user
- Password—Bind password

If you are using the sample ODBC database, you can supply the user name and password of the user used to create this database. For example, if Administrator was used to create your sample ODBC database on Windows, you can provide Administrator as bind user and its Windows password as bind password.

5. Click OK.

More information:

[Working with Ingres R3 or above](#) (see page 208)

Explore and Correlate Accounts

Perform the following procedure to explore and correlate the accounts in the ODBC-SDK endpoint.

To explore and correlate the accounts in the ODBC-SDK endpoint

1. Select the ODBC-SDK Directory object in the list view.
2. Right-click and select Explore/Correlate.
3. Complete the Explore and Correlate dialog.
4. Click Start.

ODBC-SDK Connector Code

The code for the ODBC-SDK connector includes a GUI plug-in and agent plug-in. The complete ODBC-SDK Connector code sample is located in the following directory:

PSDKHOME/Samples/ODBC-SDK

The GUI plug-in code is located in the GUI subdirectory:

PSDKHOME/Samples/ODBC-SDK/GUI

The agent plug-in code is located in the Agent subdirectory:

PSDKHOME/Samples/ODBC-SDK/Agent

The populate command code is located in the Populate subdirectory:

PSDKHOME/Samples/ODBC-SDK/Populate

Chapter 12: Common Program Exits

Common program exits let you write software to run during certain Provisioning Server actions, thereby extending the framework of the Provisioning Server with added functionality. Common program exits are called by the Provisioning Server during processing of user-provisioning operations.

Native program exits are optional exits that may be present in some connectors where such facilities are available and their use is warranted (for example the OS400 connector).

UPC program exits are part of the Universal Provisioning Connector. Both native and UPC exits are called from their respective connector plug-in running under the C++ Connector Server.

This section contains the following topics:

[Program Exit Architecture](#) (see page 211)

[Program Exit Hierarchy and Order](#) (see page 212)

[Common Program Exit Structure](#) (see page 213)

[eTExitType](#) (see page 220)

[Custom Function Program Exits](#) (see page 230)

[Sample Flow/Execution Diagram](#) (see page 233)

[Code Examples](#) (see page 233)

Program Exit Architecture

Program exits let you reference custom code from the Provisioning Server process flow. Many entry points are available where custom code can be referenced. In addition, you can invoke program exits as custom functions during policy rule evaluation so you can write custom logic to compute account attribute values. For example, to install some files on a system every time a UNIX account is created, you can write a program exit that creates the file, and indicate that the program exit be run whenever a UNIX account is created.

Program exits belong to the following types:

- *Common exits* are executed in the Provisioning Server core infrastructure.
- *Native exits* are executed in the managed endpoints. For more information about native exits, see the connector guide for the specific endpoint.

Where the program exit is handled determines which type of exit it is, not where the exit is referenced. Program exits are implemented as separate objects in the endpoint and are referenced in these objects, allowing you to define only the exits that are necessary and associate them where they need to be referenced.

The following objects reference program exits:

- Common configuration objects
- Provisioning roles
- Account templates
- Endpoints

Each object can reference multiple program exits, including multiple exits of the same type. For example, an endpoint can reference two PRE_CREATE_ACCOUNT exits.

Program Exit Hierarchy and Order

Program exits are serialized in the Provisioning Server process flow and are both hierarchical and ordered, as described below:

- In terms of hierarchy, the exits are called as referenced from the following objects in the following order:
 - Common configuration objects
 - Provisioning roles
 - Account templates
 - Endpoints
- In terms of order, in a given hierarchy exits are called in a specified order.

An operation on a global user checks for exits to be invoked in the common object and all roles to which the global user belongs. Exits referenced by the common object are invoked before exits referenced by the provisioning role. Similarly, an operation on an account checks the account templates and the endpoints to which the account belongs for exits to be invoked. Exits referenced by the account templates are invoked before exits referenced by the endpoint.

Common Program Exit Structure

Common program exits are referred to in terms of “pre” or “post” in relation to an operation that the Provisioning Server commonly performs. Program exits have a single common interface for their calling structure. This interface consists of a single argument and a single return value. The input argument is an XML buffer representation, encoded in UTF-8, of the object being acted upon combined with any custom information from the definition of the exit. The return value is status information on the result of the program exit execution as well as any documented custom information that is required for a particular exit.

There are two types of common exits:

- DLL deployed
- SOAP executable

Program Exit Input Argument

Program exits have a single interface consisting of a single input argument, which is an XML buffer. All program exits are passed to the XML buffer with the following format:

```
<eTExitInvoke eTExitType={one of the exit types}>
  <{the objectclass of the object being processed}>
    <dn>{the full DN of the object}</dn>
    <name>{the name, that is, RDN value, of the object}</name>
    <{attribute type}>{attribute value}</{attribute type}>
    ...
  </{the objectclass of the object being processed}>
  <Authentication>
    <Type> </Type>
    <User> </User>
    <Password> </Password>
  </Authentication>
</eTExitInvoke>
```

For example:

```
<eTExitInvoke eTExitType=PRE_ADD_ACCOUNT>
  <eTSDKAccount>
    <dn>eTSDKAccountName=test1, eTSDKAccountContainerName=SDK Accounts,
      eTSDKDirectoryName=Team1, dc=Dev</dn>
    <name>test1</name>
    <eTSDKCity>Renton</eTSDKCity>
  </eTSDKAccount>
  <Authentication>
    <Type>GLOBAL_USER</Type>
    <User>{the DN of the global user}</User>
    <Password>{the password of the global user}</Password>
  </Authentication>
</eTExitInvoke>
```

For modify operations, the modify mode is specific in each tag. The possible modify modes are ADD, DELETE, and REPLACE. For example:

```
<eTExitInvoke eTExitType=PRE_MODIFY_ACCOUNT>
  <eTSDKAccount>
    <dn>eTSDKAccountName=test1, eTSDKAccountContainerName=SDK Accounts,
      eTSDKDirectoryName=Team1, dc=Dev</dn>
    <name>test1</name>
    <eTSDKCity modify-mode="replace">Kirkland</eTSDKCity>
    <eTSDKDescription modify-mode="delete">
      Old description</eTSDKDescription>
  </eTSDKAccount>
</eTExitInvoke>
```

The program exit parses this input argument to get the data it needs to perform its specific task.

If a program exit is defined to handle only a specific type of exit, it should check the eTExitType to make sure that it can handle that specific type. For example, if a program exit is designed to handle exit type PRE_ADD_ACCOUNT, it should check eTExitType and perform only its task, if the exit type is correct. If the exit type is not handled by the program exit, it should do nothing and return a warning.

Input XML Buffer Authentication Type

Each input XML buffer may contain an optional authentication XML block. The format of the authentication XML block is always defined as follows.

```
<Authentication>
  <Type> </Type>
  <User> </User>
  <Password> </Password>
</Authentication>
```

The data in the authentication XML block depends on the type of authentication defined for the program exit. The following are the possible authentication types:

NONE

No credentials are passed to the method being invoked. Thus, the input XML buffer does not contain an authentication block.

GLOBAL_USER

The credentials of the currently logged on global user are passed to the program exit being invoked. The <User> tag contains the DN of the global user. The <Password> tag contains the password for that global user.

Note: The password is not encrypted.

PROXY

The credentials of a specific global user are passed to the program exit being invoked. The <User> tag contains the DN of the specific global user. The <Password> tag contains the password for that global user.

Note: The password is not encrypted.

OTHER

Indicates that the <User> and <Password> tags are program-exit specific. The <User> and <Password> tags can be any free form text. It is up to the program exits to define what these fields mean.

Program Exit Return Value

Program exits have a single return value, which is an XML buffer. Program exits must return an XML buffer, which has the following format:

```
<eTExitReturn>
  <eTExitReturnCategory> </eTExitReturnCategory>
  <eTExitReturnNative> </eTExitReturnNative>
  <eTExitLogMsg> </eTExitLogMsg>
  <eTExitContinue> </eTExitContinue>
  <eTExitCustom> </eTExitCustom>
  <eTPersistentFailure> </eTPersistentFailure>
</eTExitReturn>
```

eTExitReturnCategory XML

Requirement

This value is not required.

Purpose

Groups various native return codes into one of three categories for the purpose of simplifying process flow.

Valid Values

SUCCESS

WARNING

FAILURE

Default Values

If no value is specified, SUCCESS is assumed.

eTExitReturnNative XML

Requirement

This value is not required.

Purpose

Specifies the return value from the native program exit call.

Valid Values

This value is a string representation of what occurred.

Default Values

None.

eTExitLogMsg XML**Requirement**

This value is not required. It is, however, highly recommended to enter a value for failure or warning responses:

- Without eTExitLogMsg value, the server will send the eTExitReturnNative code for logging.
- Without eTExitLogMsg and eTExitReturnNative values, the server will make up a generic message indicating no message present and that there was an error/ warning.

Purpose

Specifies a string value that the native program exit wants the server to log.

Valid Values

This value will be a UTF-8 string.

Default Values

None.

eTExitContinue XML**Requirement**

This value is not required.

Purpose

Specifies whether to continue the process flow after the return from the program exit. This value overrides the default behavior. See Default Values.

Valid Values

TRUE - Continue Execution.

FALSE - Stop Execution.

Default Values

The default values are based on the eTExitReturnCategory attribute.

TRUE - If eTExitReturnCategory is SUCCESS or WARNING.

FALSE - If eTExitReturnCategory is FAILURE.

eTExitCustom XML

Requirement

This value is not required.

Purpose

For common program exits, this value is reserved for future use.

For native exits, this value is connector-specific.

Valid Values

Any valid XML document.

Default Values

None.

The program exit parses this input argument to get the data it needs to perform its specific task.

eTPersistentFailure

Requirement

This value is not required.

Purpose

Used only in responses from IMS Notifications, which share with program exits the same XML buffers for encoding requests and responses. A persistent failure is a notification that is rejected based on a problem in the content (likely a programming error) rather than based on some retry-able situation.

Valid Values

TRUE - Indicates a persistent failure.

FALSE - Indicates a transient failure, one that might succeed later if retried.

Default Values

FALSE

Common Exits DLL Interface

DLL deployed program exits must export the function with the following prototype:

```
int function_name(  
    char Input_XML,  
    char *      Return_XML,  
    int *Return_Buffer_Length)
```

The following list describes the parameters for the DLL deployed program exit prototype:

function_name

Name of the program exit. One DLL can export multiple program exits, where each program exit is an exported function with the prototype defined above.

InputXML

Character buffer in UTF-8 format. It contains the XML buffer that the Provisioning Server passes to the program exit.

ReturnXML

Character buffer in UTF-8 format. It is an empty buffer that the Provisioning Server passes to the program exit for it to send a return value back to the Provisioning Server. The size of the buffer is passed to the program exit is the Return_Buffer_Length parameter.

Return_Buffer_Length

Both an input and output parameter:

- On input, Return_Buffer_Length indicates the maximum length, in characters, that the Return_XML buffer can contain. The program exits must not exceed this length when building the return XML buffer.
- On output, Return_Buffer_Length contains the actual length of the return XML buffer the program exit built. That is, after the program exit builds the return XML buffer, it sets Return_Buffer_Length to the actual length of the buffer being returned.

Common Exits SOAP Interface

SOAP-deployed program exits must present an external interface like the following prototype:

```
char * function_name ( char * Input_XML )
```

The following list describes the parameters for the SOAP-deployed program exit prototype:

function_name

Name of the program exit. The Web Services Description Language (WSDL) file that describes the program exit contains a definition of the `function_name`. This is also the name of the character buffer in UTF-8 format that is returned. This buffer is allocated by the referenced program, but must be cleared from the calling program.

Input_XML

Character buffer in UTF-8 format. `Input_XML` contains the XML buffer the Provisioning Server passes to the program exit.

The definition of this interface needs to be presented in the following ways:

- A way that the SOAP client can understand.
- A way that the SOAP server can understand.

The SOAP client relies upon WSDL to specify the interface. For a description of WSDL, see <http://www.w3.org/tr/wsdl.html>.

The SOAP server described here is the Apache SOAP server. The Apache SOAP server requires an XML document known as a Deployment Descriptor. The Deployment Descriptor indicates to the SOAP server what the interface to the SOAP program is. For a more complete description of deployment descriptors, see the Deployment Descriptors section in the *User Guide* at <http://ws.apache.org/soap/docs/index.html>.

An example of a SOAP exit can be found in the following folder:

`Samples/ProgramExitSOAP`

eTExitType

Exit types determine the circumstances under which an exit is called. A value is entered for `eTExitType`, in the input XML buffer passed to the program exit.

The exit types with `ACCOUNT` in their names can be common or native exits, meaning that common code and connector code can be triggered to process them.

All other exit types, however, must be common exits. It should also be noted that not all program exit types are referenced from the various object types.

Notes:

- In all cases, the name of the object being passed is sent. This is formatted in both DN and Common Name format.
- To have complete control over passwords, either at the global user or the account levels, you must provide exits both for create user/account and for change password user/account. In other words, for new global users (accounts), the change password exit is not called. For new global users (accounts), the password is passed in as part of the attribute for the create exit (for example, PRE_CREATE_GLOBAL_USER).

More information:

[Valid Values for eTExitType](#) (see page 221)

Valid Values for eTExitType

The following values are valid for eTExitType:

PRE_ADD_ACCOUNT

The account information that is being passed to the create account request is also passed to this program. Unlike the Modify operation, the password is passed to the Create operation as part of the account information.

POST_ADD_ACCOUNT

The account information that is being passed to the create account request is also passed to this program. Unlike the Modify operation, the password is passed to the Create operation as part of the account information.

PRE_MODIFY_ACCOUNT

The account information that is being passed to the modify account request is also passed to this program. The only exclusion to this is the password attribute.

POST_MODIFY_ACCOUNT

The account information that is being passed to the modify account request is also passed to this program. The only exclusion to this is the password attribute.

PRE_CHANGE_ACCOUNT_PASSWORD

A special case of MODIFY. This exit is triggered when the password attribute for the account changes. If the password attribute is the only change, the other modify code is not triggered. If other attributes change, this code is triggered. The account name and the password attributes contain the only information available to this program exit.

POST_CHANGE_ACCOUNT_PASSWORD

A special case of MODIFY. This exit is triggered when the password attribute for the account changes. If the password attribute is the only change, the other modify code is not triggered. If other attributes change, this code is triggered. The account name and the password attribute contain the only information available to this program exit.

PRE_ENABLE_ACCOUNT

A special case of MODIFY. This exit is triggered when the enable attribute for the account changes. If the enable attribute is the only change, the other modify code is not triggered. If other attributes change, this is triggered. The account name is the only account attribute available to this exit.

POST_ENABLE_ACCOUNT

A special case of MODIFY. This exit is triggered when the enable attribute for the account changes. If the enable attribute is the only change, the other modify code is not triggered. If other attributes change, this is triggered. The account name is the only account attribute available to this exit.

PRE_DISABLE_ACCOUNT

A special case of MODIFY. This exit is triggered when the disable attribute for the account changes. If the disable attribute is the only change, the other modify code is not triggered. If other attributes change, this is triggered. The account name is the only account attribute available to this exit.

POST_DISABLE_ACCOUNT

A special case of MODIFY. This exit is triggered when the disable attribute for the account changes. If the disable attribute is the only change, the other modify code is not triggered. If other attributes change, this is triggered. The account name is the only account attribute available to this exit.

PRE_DELETE_ACCOUNT

Triggered prior to a DELETE request. The account name is the only account attribute available to this exit.

POST_DELETE_ACCOUNT

Triggered after a DELETE request. The account name is the only account attribute available to this exit.

PRE_ADD_GLOBAL_USER

The global user information that is being passed to the create request is also passed to this program.

POST_ADD_GLOBAL_USER

The global user information that is being passed to the create request is also passed to this program.

PRE_MODIFY_GLOBAL_USER

The global user information that is being passed to the modify request is also passed to this program. The only exclusion to this is the password attribute.

POST_MODIFY_GLOBAL_USER

The global user information that is being passed to the modify request is also passed to this program. The only exclusion to this is the password attribute.

PRE_CHANGE_GLOBAL_USER_PWD

A special case of MODIFY. This exit is triggered when the password attribute for the global user changes. If the password attribute is the only change, the other modify code is not triggered. If other attributes change, this code is triggered. The global user name, the password, and optionally the password clue attributes are the only information available to this program exit.

POST_CHANGE_GLOBAL_USER_PWD

A special case of MODIFY. This exit is triggered when the password attribute for the global user changes. If the password attribute is the only change, the other modify code is not triggered. If other attributes change, this code is triggered. The global user name, the password, and optionally the password clue attributes are the only information available to this program exit.

PRE_ENABLE_GLOBAL_USER

A special case of MODIFY. This exit is triggered when the enable attribute for the global user changes. If the enable attribute is the only change, the other modify code is not triggered. If other attributes change, this code is triggered. The global user name is the only attribute available to this exit.

POST_ENABLE_GLOBAL_USER

A special case of MODIFY. This exit is triggered when the enable attribute for the global user changes. If the enable attribute is the only change, the other modify code is not triggered. If other attributes change, this code is triggered. The global user name is the only attribute available to this exit.

PRE_DISABLE_GLOBAL_USER

A special case of MODIFY. This exit is triggered when the disable attribute for the global user changes. If the disable attribute is the only change, the other modify code is not triggered. If other attributes change, this code is triggered. The global user name is the only attribute available to this exit.

POST_DISABLE_GLOBAL_USER

A special case of MODIFY. This exit is triggered when the disable attribute for the global user changes. If the disable attribute is the only change, the other modify code is not triggered. If other attributes change, this code is triggered. The global user name is the only attribute available to this exit.

PRE_DELETE_GLOBAL_USER

Triggered prior to a DELETE request. The global user name is the only attribute available to this exit.

POST_DELETE_GLOBAL_USER

Triggered after a DELETE request. The global user name is the only attribute available to this exit.

PRE_ASSOCIATE_ROLE

Refers to the changing of provisioning role membership in the Provisioning Server, regardless of what happens at the account level. The global user name and the provisioning role name is the only information available to this program exit.

POST_ASSOCIATE_ROLE

Refers to the changing of provisioning role membership in the Provisioning Server, regardless of what happens at the account level. The global user name and the provisioning role name is the only information available to this program exit.

PRE_DISASSOCIATE_ROLE

Refers to the changing of provisioning role membership in the Provisioning Server, regardless of what happens at the account level. The global user name and the provisioning role name are the only information available to this program exit.

Note: This value is called only for incremental provisioning role changes. If you use a replace-mode modification of global user's provisioning roles to replace one set of provisioning roles with another, this value calls the associate-role exits only. The exit would not read the database to find out which provisioning roles were previously included to see which were being set that were previously set and which were being removed.

POST_DISASSOCIATE_ROLE

Refers to the changing of provisioning role membership in the Provisioning Server, regardless of what happens at the account level. The global user name and the provisioning role name are the only information that is available to this program exit.

Note: This value is called only for incremental provisioning role changes. If one uses a replace-mode modification of global user's provisioning roles to replace one set of provisioning roles with another, this value calls the associate-role exits only. The exit would not read the database to find out which provisioning roles were previously included to see which were being set that were previously set and which were being removed.

PRE_ADD_GLOBAL_GROUP

The global group information that is being passed to the add request is also passed to this program.

POST_ADD_GLOBAL_GROUP

The global group information that is being passed to the add request is also passed to this program.

PRE_MODIFY_GLOBAL_GROUP

The global group information that is being passed to the modify request is also passed to this program.

POST_MODIFY_GLOBAL_GROUP

The global group information that is being passed to the modify request is also passed to this program.

PRE_DELETE_GLOBAL_GROUP

Triggered prior to a delete request. The global group name is the only attribute available to this exit.

POST_DELETE_GLOBAL_GROUP

Triggered after a delete request. The global group name is the only attribute available to this exit.

PRE_ADD_ROLE

The provisioning role information that is being passed to the add request is also passed to this program.

POST_ADD_ROLE

The provisioning role information that is being passed to the add request is also passed to this program.

PRE_MODIFY_ROLE

The provisioning role information that is being passed to the modify request is also passed to this program.

POST_MODIFY_ROLE

The provisioning role information that is being passed to the modify request is also passed to this program.

PRE_DELETE_ROLE

Triggered prior to a delete request. The provisioning role name is the only attribute available to this exit.

POST_DELETE_ROLE

Triggered after a delete request. The provisioning role name is the only attribute available to this exit.

CUSTOM_FUNCTION

Triggered when a program exit is invoked through a policy rule expression such as %\$funcname(%UN%,%AC%)%.

More information:

[Custom Function Program Exits](#) (see page 230)

PMC_VALIDATE_PASSWORD

This value is triggered during the change of a global user or account password when the Provisioning Server is configured, by the Identity Manager Server/Use External Password Policies parameter, to perform certain password validation through the Identity Manager Server instead of by using the Provisioning Server password profile object. For a description of this configuration parameter, see the *CA Identity Manager Provisioning Guide*.

PMC_UPDATE_PASSWORD_HISTORY

This value is triggered after the completion of a global user password change when the Provisioning Server is configured, by the Identity Manager Server/Use External Password Policies parameter, to perform password validation through the Identity Manager Server instead of by using the Provisioning Server password profile object. For a description of this configuration parameter, see the *CA Identity Manager Provisioning Guide*.

Containment

Containment refers to the allowed combination of objects and program reference type, and not to X.500 containment.

Common Configuration Object

The Common Configuration Object is used to assign program exits for the global user, global user group, or provisioning role object classes. For example, if certain program exits should be called when global users are processed, the common configuration object should reference those exits. In addition, the common configuration object is needed to provide a way to call exits during the add operation.

- PRE_ADD_GLOBAL_USER
- POST_ADD_GLOBAL_USER
- PRE_MODIFY_GLOBAL_USER
- POST_MODIFY_GLOBAL_USER
- PRE_CHANGE_GLOBAL_USER_PWD
- POST_CHANGE_GLOBAL_USER_PWD
- PRE_ENABLE_GLOBAL_USER
- POST_ENABLE_GLOBAL_USER
- PRE_DISABLE_GLOBAL_USER
- POST_DISABLE_GLOBAL_USER
- PRE_DELETE_GLOBAL_USER
- POST_DELETE_GLOBAL_USER
- PRE_ADD_GLOBAL_GROUP
- POST_ADD_GLOBAL_GROUP
- PRE_MODIFY_GLOBAL_GROUP
- POST_MODIFY_GLOBAL_GROUP
- PRE_DELETE_GLOBAL_GROUP
- POST_DELETE_GLOBAL_GROUP
- PRE_ADD_ROLE
- POST_ADD_ROLE
- PRE_MODIFY_ROLE
- POST_MODIFY_ROLE
- PRE_DELETE_ROLE
- POST_DELETE_ROLE
- PMC_VALIDATE_PASSWORD
- PMC_UPDATE_PASSWORD_HISTORY

Provisioning Roles

A role object can reference program exits to assign the exits that are invoked for various operations on global users associated with that provisioning role. If a provisioning role references a program exit, those exits are called in addition to the exits referenced by the Common Configuration Object. The exits defined on the common configuration object are invoked before exits defined on the provisioning role (hierarchy order).

When adding a global user (PRE_ADD_GLOBAL_USER and POST_ADD_GLOBAL_USER exit types), program exits are invoked based on the initial set of provisioning roles being assigned to the user.

When associating or disassociating a provisioning role with a global user (PRE_ASSOCIATE_ROLE, POST_ASSOCIATE_ROLE, PRE_DISASSOCIATE_ROLE and POST_DISASSOCIATE_ROLE exit types), the program exits referenced by the provisioning roles being associated or disassociated are invoked.

When modifying an existing global user in other ways (using the exit types listed below), all provisioning roles to which the global user belongs are consulted to identify program exits to invoke.

Other Exit Types

The common configuration object handles add exits for a global user.

- PRE_ADD_GLOBAL_USER
- POST_ADD_GLOBAL_USER
- PRE_ASSOCIATE_ROLE
- POST_ASSOCIATE_ROLE
- PRE_DISASSOCIATE_ROLE
- POST_DISASSOCIATE_ROLE
- PRE_MODIFY_GLOBAL_USER
- POST_MODIFY_GLOBAL_USER
- PRE_CHANGE_GLOBAL_USER_PWD
- POST_CHANGE_GLOBAL_USER_PWD
- PRE_ENABLE_GLOBAL_USER
- POST_ENABLE_GLOBAL_USER

- PRE_DISABLE_GLOBAL_USER
- POST_DISABLE_GLOBAL_USER
- PRE_DELETE_GLOBAL_USER
- POST_DELETE_GLOBAL_USER

Account Templates

An account template object can reference program exits to affect the accounts associated with that template.

If an account template references program exits, these exits are called in addition to the exits that are referenced by the endpoint to which the account belongs. The exits defined on the account template are invoked before the exits on the endpoint (hierarchy order).

If an account is being created from one or more account templates (PRE_ADD_ACCOUNT and POST_ADD_ACCOUNT exit types), those template exits are called.

When working with an existing account, whether the current set of assigned account templates is being adjusted or not, it is the initial set of assigned templates whose program exits are invoked.

- PRE_ADD_ACCOUNT
- POST_ADD_ACCOUNT
- PRE_MODIFY_ACCOUNT
- POST_MODIFY_ACCOUNT
- PRE_CHANGE_ACCOUNT_PASSWORD
- POST_CHANGE_ACCOUNT_PASSWORD
- PRE_ENABLE_ACCOUNT
- POST_ENABLE_ACCOUNT
- PRE_DISABLE_ACCOUNT
- POST_DISABLE_ACCOUNT
- PRE_DELETE_ACCOUNT
- POST_DELETE_ACCOUNT

Endpoints

Endpoint objects are used to assign program exits to accounts. For example, if certain program exits should be called when accounts are processed, the endpoint objects should reference those exits.

In addition, endpoint objects are needed to provide a way to call exits during an add operation.

- PRE_ADD_ACCOUNT
- POST_ADD_ACCOUNT
- PRE_MODIFY_ACCOUNT
- POST_MODIFY_ACCOUNT
- PRE_CHANGE_ACCOUNT_PASSWORD
- POST_CHANGE_ACCOUNT_PASSWORD
- PRE_ENABLE_ACCOUNT
- POST_ENABLE_ACCOUNT
- PRE_DISABLE_ACCOUNT
- POST_DISABLE_ACCOUNT
- PRE_DELETE_ACCOUNT
- POST_DELETE_ACCOUNT

Custom Function Program Exits

A custom function program exit is invoked from a policy rule expression. For details on setting custom rule function invocations as policy attribute values, see the *CA Identity Manager Provisioning Guide*. Custom function program exits share the following characteristics:

- The exit type is always CUSTOM_FUNCTION. There are no PRE or POST variants.
- The exit must be a common program exit (DLL or SOAP). Native exits cannot be used to compute the custom function.
- The exit must be registered in the same domain as the account being created or updated from the policy. The reference to a custom function program exit (%\$funcname(...)%) contains the name of the exit (funcname), but there is no rule string syntax to let you specify the domain of the program exit so it is always presumed to be in the domain of the account.
- The input to the program exit includes zero or more single- or multi-valued parameters.

For example, if the global user for the account being created or updated has the following attribute settings:

eTCustomField01: { value1a, value1b } (that is, two values assigned)
eTCustomField02: value2

the `%%*/$FuncName(%*UCU01%, %UCU02%)%` rule expression is evaluated.

The input XML passes all values of eTCustomField01 and the value of eTCustomField02 as follows:

```
<eTExitInvoke eTExitType=CUSTOM_FUNCTION>
  <eTFunction>
    <eTFuncParam1>value1a</eTFuncParam1>
    <eTFuncParam1>value1b</eTFuncParam1>
    <eTFuncParam2>value2</eTFuncParam2>
  </eTFunction>
  <Authentication>
    <Type>GLOBAL_USER</Type>
    <User>{the DN of the global user}</User>
    <Password>{the password of the global user}</Password>
  </Authentication>
</eTExitInvoke>
```

The output from the program exit can indicate an error (as with any other program exit) so that the creation or update of the account is not attempted, or can contain a single- or multi-valued output parameter. For example, a program exit could return the following XML block to indicate two values (ReturnValue1 and ReturnValue2) to set for the corresponding account attribute:

```
<eTExitReturn>
  <eTExitReturnCategory>SUCCESS</eTExitReturnCategory>
  <eTExitReturnNative>0</eTExitReturnNative>
  <eTExitContinue>TRUE</eTExitContinue>
  <eTExitCustom>
    <eTFuncReturn>ReturnValue1</eTFuncReturn>
    <eTFuncReturn>ReturnValue2</eTFuncReturn>
  </eTExitCustom>
</eTExitReturn>
```

The function rule expression controls the number of values to set as follows:

- If the `$FuncName` in the rule expression is preceded by `*` (asterisk) as in the example above, this will set 0 or more values of the attribute depending on what is included in the output XML document.
- If the function rule expression does not have the `*` preceding `$FuncName`, only the first value returned is relevant. Additional values are ignored.

Obscured Returned Values

The program exit returns information inside the eTFuncReturn XML block, for example:

```
<eTFuncReturn>Returned value from program exit</eTFuncReturn>
```

If logging is enabled, then this XML block can be read.

However, if the program exit returns information like a password, then you may not want the information to be logged. In this case, you can flag the returned value as obscured to prevent it from being logged.

The format of the obscured value is:

```
<eTFuncReturn obscured="yes">MyPassword</eTFuncReturn>
```

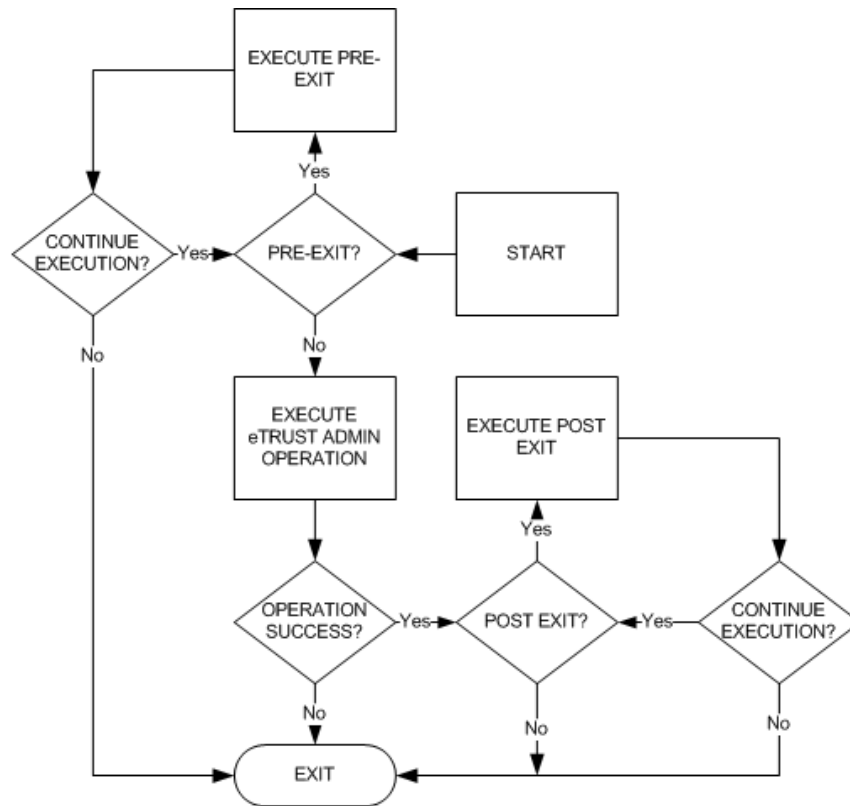
This tells the Provisioning Server to replace the value with the string **** NOT SHOWN **** as it does for attribute names that are recognized as storing sensitive attributes. For example:

```
<eTFuncReturn obscured="yes">** NOT SHOWN **</eTFuncReturn>
```

Note: The obscured attribute is case-sensitive. For example, the result will not be replaced correctly if the attribute is set to "YES".

Sample Flow/Execution Diagram

The following illustration provides a sample flowchart of the execution logic of a program exit.



Code Examples

Use the code examples located in the following directory as a guide when coding your Common Program Exits:

Samples\ProgramExits
 Samples\ProgramExitSOAP

Chapter 13: Universal Provisioning Program Exits

The Universal Provisioning Connector (UPC) provides its own Program Exit functionality, which is based on the Provisioning Server Common Exits and not on the endpoint's Native Exit approach.

Universal Provisioning Program exits let you write software that executes user-specified code for specific Provisioning Server actions, like the Common Program exits.

UPC exits extend the framework of the Provisioning Server, allowing for additional functionality that can change or augment the standard functionality.

The major difference between Common and UPC exits is that UPC exits allow for the execution of a user-defined program for the actual user-provisioning operation. The second difference is that, at your option, user-specified data can be added to the XML payload by specifying it through the UPC Program Exit object or through the UPC Exit Reference. This user data is transparent to the Provisioning Server and is intended for processing by the program exit itself.

More information:

[Using Program Exits with UPC Example](#) (see page 253)

This section contains the following topics:

[UPC Program Exit Architecture](#) (see page 236)

[Implementing UPC Program Exits in Multiple Domain Environments](#) (see page 237)

[UPC Program Exit Structure](#) (see page 238)

[eTExitType](#) (see page 246)

[Sample Flow/Execution Diagrams](#) (see page 249)

[Code Examples](#) (see page 252)

UPC Program Exit Architecture

UPC Program exits let you execute custom code when the Provisioning Server process flow requires user provisioning operations. UPC can operate in one of the following operational modes: non-managed or managed mode.

Non-managed mode

In non-managed mode, program exits are called asynchronously. When the exit is invoked, it returns the status of the exit being invoked and not the status of the user-provisioning operation itself.

Managed mode

In managed mode, program exits are called synchronously. The exit must perform the user-provisioning operation, and return the status of this operation.

Entry Points in Non-Managed Mode

The following entry points are available:

- Seven for user-provisioning actions
- Two for flagging pending and completed user-provisioning requests
- Three for service-level-agreements support
- One to flag a program exit invocation error

The non-managed operation mode of the Universal Provisioning Connector is designed to integrate non-managed systems into the overall Provisioning Server user-provisioning framework.

The UPOSendMail exit provides a program to email the remote system administrator to request user-provisioning actions.

Entry Points in Managed Mode

Six entry points are available for user-provisioning actions, and another one is available to flag a program exit invocation error.

UPC Program Exit Objects

UPC Program exits are implemented as separate objects in the UPC endpoint and are referenced in these objects, letting you define as few exits as are necessary, and associate them where they need to be referenced. Program exits are referenced by UPC Accounts, by the UPC Service Monitor, and by the Exit invocation itself.

Each of these can reference multiple program exits, including multiple exits of the same type.

Program Exit Order

Program exits are serialized in the Provisioning Server process flow by their assigned priority level.

Implementing UPC Program Exits in Multiple Domain Environments

If you are managing global users in a multiple domain environment, UPC program exit processing can take place in the following locations:

- The domain where the UPC Account object is being processed
- The domain where the UPC program exit reference is defined
- The domain where the UPC program exit is defined

If the domain where the object is being processed is different from the domain where the program exit is defined, the global user who initiates the processing must have MODIFY access to the program exit in the domain where the exit is defined.

If the global user does not have MODIFY access to the program exit object in the domain where it is defined, then the program exit will fail. This may happen if program exits are invoked by self-administrators since they typically do not have MODIFY access.

Note: If you are using program exits in a multiple domain environment, you must make sure global users have the necessary access to objects in other domains.

Note: The default UPC administrator, etapoad, has been assigned sufficient rights for the domain in which UPC is installed.

UPC Program Exit Structure

Unlike Common Program exits, which are referred to in terms of “pre-” or “post-” operation, UPC Program exits are referred to by the name of the operation itself. Program exits have a single common interface for their calling structure. This interface consists of a single argument and a single return value. The argument is an XML buffer representation, encoded in UTF-8, of the object being acted upon (a UPC Account), any custom information from the definition of the exit, and if operating in non-managed mode, details of the user-provisioning operation.

The interpretation of the return value depends on the operating mode.

- In non-managed mode, the return value is the status information about the result of the program exit execution as well as any documented custom information that is required for a particular exit.
- In managed mode, the return value is the status information of the user-provisioning operation itself, as well as any data that needs to be returned to the Provisioning Server.

UPC exits belong to one of the following types:

- DLL deployed
- SOAP executable

Input XML Buffer (Input Argument)

Program exits have a single interface consisting of a single input argument, which is an XML buffer. All program exits are passed to the XML buffer with the following format:

```
<eTExitInvoke eTExitType="{one of the exit types}">

  <eTUP00operation>
    <{attribute type}>{attribute value}</{attribute type}>
    ...
  </eTUP00operation>

  <eTUP0Account>
    <eTDN> {the full DN of the object} </eTDN>
    <eTName>{the name, that is, RDN value, of the object}</eTName>
    <{attribute type}>{attribute value}</{attribute type}>
    ...
  </eTUP0Account>
```

```
<Authentication>
  <Type> </Type>
  <User> </User>
  <Password> </Password>
</Authentication>

<eTExitCustomData>
  ...
</eTExitCustomData>
  </eTExitInvoke>
```

The program exit parses this input argument to get the data it needs to perform its specific task.

If a program exit is defined to handle only a specific type of exit, it should check the eTExitType to make sure that it can handle that specific type. For example, if a program exit is designed to handle exit type ADD_ACCOUNT, it should check eTExitType and perform its task only if the exit type is correct. If the exit type is not handled by the program exit, it should do nothing and return a warning.

Input XML Buffer Operation Block

This block is only available when UPC is configured to operate using non-managed mode asynchronous exits.

The Operation block provides a wrapper for elements describing the user-provisioning request associated with the account being processed.

The format of this block is as follows:

```
<eTUP00peration>
  <eTUP00AccountStatus> </eTUP00AccountStatus>
  <eTUP00operationType> </eTUP00operationType>
  <eTUP00operationID> </eTUP00operationID>
  <eTUP00operationDate> </eTUP00operationDate>
  <eTUP00operationTime> </eTUP00operationTime>
  <eTUP00operationTimeStamp> </eTUP00operationTimeStamp>
  <eTUP00RootOperationID> </eTUP00RootOperationID>
  <eTPolicyDN> </eTPolicyDN>
</eTUP00peration>
```

The content of the operation XML block is as follows:

- **UPC Account Status**-Indicates the pre-operation status of the account. The value is a single letter:
 - A—Active account
 - D—Deleted account
 - P—Account that has an operation still pending against it.
 - R—Account marked for physical removal from the repository upon the next receipt of a DELETE request.
- **Operation Type**-Indicates the operation type. The value is a single integer:
 - 0—Add operation
 - 1—Delete operation
 - 2—Modify operation, including password and enable/disable operations
 - 3—Rename operation
- **Operation ID**-A string generated by the UPC Agent to uniquely identify the current operation. The format is as generated by Microsoft's Platform SDK: Remote Procedure Call (RPC) UuidToString() function.
- **Operation Date**-The date of the operation, in Unicenter Date format.
- **Operation Time**-The time of the operation, in Unicenter Time format.
- **Operation TimeStamp**-The date and time of the operation using the time_t format.
- **Root Operation ID**-(ETA) The Provisioning Server ID of the top-level operation which called upon UPC for a user-provisioning operation. The format is as generated by Microsoft's Platform SDK: Remote Procedure Call (RPC) UuidToString() function.

Note: This element sometimes is present in the XML Account block instead.
- **Policy DN**-(ETA) The distinguished name of the policy associated with this account.

Note: This element sometimes is present in the XML Account block instead.

Input XML Buffer Account Block

The account block is an XML representation of all the attribute/value pairs of the account object provided to the UPC Agent by the superagent server plug-in.

The format of the XML account block is as follows:

```
<eTUPOAccount>
  <eTDN> </eTDN>
  <eTName> </eTName>
  <eTUPOAccountName> </eTUPOAccountName>
  <objectClass>eTUPOAccount</objectClass>
  <eTPassword> </eTPassword>
  <eTUPOUserData> </eTUPOUserData>
  <eTAccountStatus>A</eTAccountStatus>
  <eTSuspended>0</eTSuspended>
  <eTPolicyDN> </eTPolicyDN>
  <eTUPORootOperationID> </eTUPORootOperationID>
  <creatorsName>CN=ROOT,DC=ETASA</creatorsName>
  <createTimestamp>20041126185959Z</createTimestamp>
  <modifiersName>CN=ROOT,DC=ETASA</modifiersName>
  <modifyTimestamp>20041126185959Z</modifyTimestamp>
  <eTNewDN> </eTNewDN>
  <eTNewName> </eTNewName>
</eTUPOAccount>
```

The content of the account XML block is as follows:

Note: Not all the elements may be present.

- **DN**—(ETA) Usually, the full distinguished name of the account. For some exits, such as the LIST_ACCOUNTS type, the DN will have the value of the account container.
- **Name**—(ETA) Usually, the value of the naming attribute for the associated account. For some exits, such as the LIST_ACCOUNTS type, the name is the naming value of the account container.
- **UPC Account Name**—(UPC) Same as Name.
- **Account Status**—(ETA) attribute indicating the status of the account. The value is a single letter:
 - A—Active account
 - S—Suspended account
 - I—Inactive account.

- **Suspended**—(ETA) attribute showing the suspension status. The value is a single integer:
 - 0—Active
 - 1—Suspended
 - 2—Manual
- **Root Operation ID**—(ETA) The Provisioning Server ID of the top-level operation which called upon UPC for a user-provisioning operation. The format is as generated by Microsoft's Platform SDK: Remote Procedure Call (RPC) UuidToString() function.

Note: This element sometimes is present in the XML Operation block instead.
- **Policy DN**—(ETA) The full distinguished name of the policy associated with this account.

Note: This element sometimes is present in the XML Operation block instead.
- **Object Class**—(ETA) The LDAP object class of the object will always be eTUPOAccount.
- **Password**—(ETA) The password to be assigned to the account. The password is in the clear and not encrypted.

Note: When in non-managed mode, we strongly recommend that you not manage user passwords with UPC because they can be easily compromised. It is especially important not to use the %P% rule-string in UPC policies as this would permit the global user's password to be compromised.
- **NewDN**—(UPC) Only for the RENAME_ACCOUNT exit. The new full distinguished name of the account.
- **newName**—(UPC) Only for the RENAME_ACCOUNT exit. The new name for the associated account.

The following element is from the UPC Account schema and conveys information about the user to be passed to the non-managed system:

- **UPC User Data**—(UPC) User-specified data. Specifies the format to use when populating this attribute as the data may need to be parsed by the exit that will be invoked. The data is transparent and, thus, not processed by the Provisioning Server.

Input XML Buffer Authentication Type

UPC supports the same authentication type as Common Exits.

Each input XML buffer may contain an optional authentication XML block. The format of the authentication XML block is always defined as follows.

```
<Authentication>
  <Type> </Type>
  <User> </User>
  <Password> </Password>
</Authentication>
```

The data in the authentication XML block depends on the type of authentication defined for the program exit. The following are the possible authentication types:

NONE

Indicates that no credentials are passed to the method being invoked. Thus, the input XML buffer does not contain an authentication block.

GLOBAL_USER

Indicates that the currently logged on global user's credentials, when available, are passed to the program exit being invoked. The <User> tag contains the DN of the global user. The <Password> tag contains the password for that global user. If the current global user's credentials are not available, the credentials of the global user used to connect to the Provisioning Server are used. That is, the credentials configured in the Directory property sheet or through the UPC Monitor Service command line interface are used.

Note: The password is in the clear.

PROXY_USER

Indicates that a specific global user's credentials are passed to the program exit being invoked. The <User> tag contains the DN of the specific global user. The <Password> tag contains the password for that global user.

Note: The password will be in the clear.

PROXY_OTHER

Indicates that the <User> and <Password> tags are program exit specific. The <User> and <Password> tags can be any free form text. It is up to the program exits to define what these fields mean.

XML Exit Custom Data Block

The Exit Custom Data block is specific to UPC exits. It provides a facility to pass along user-specified data to program exits. The custom data block has the following form:

```
<eTExitCustomData>
  <someTag>
    <value>value</value>
  </someTag>
  <IgnoreFailure>FALSE</IgnoreFailure>
  <sampleTag>
    <value1>value1</value1>
  </sampleTag>
</eTExitCustomData>
```

<someTag> and *<sampleTag>* blocks and their content are user defined, as shown in the following examples:

- **<someTag>**—XML block defined in the CustomData field of the eTUPOExitReference attribute of the UPC Directory object.
- **<sampleTag>**—XML block defined in the eTUPOExitCustomData attribute of the UPC Program Exit object.

The following custom data blocks have been defined for UPC:

- **<IgnoreFailure>**—This element is used by the Exit invocation routines to determine if on an exit failure, the INVOCATION_ERROR exit should be invoked. The value is a text representation of a boolean, either TRUE or FALSE.

- **<eTUPOSendMail>**—The SendMail block is used by the non-managed mode UPC SendMail exit to provide all the necessary details to submit SMTP messages to one or more recipients. The block has the following format:

```
<eTUPOSendMail>
  <MailHost>smtphost.test.com</MailHost>
  <MailFrom>fromoriginator@test.com</MailFrom>
  <MailPort>25</MailPort>
  <MailTLS>TRUE</MailTLS>
  <MailAuth>TRUE</MailAuth>
  <MailUser>smtpuser</MailUser>
  <MailPassword>
    <TEncrypted>{V01}E2hfp91UGFgcJMdpT2F8Gw==</TEncrypted>
  </MailPassword>
  <MailTo>torecipient@test.com</MailTo>
  <MailCc>ccrecipient@test.com</MailCc>
  <MailSubject>subject line</MailSubject>
  <MailBody>sample body text</MailBody>
</eTUPOSendMail>
```

Note: If an element is defined in both the Exit object and the Exit Reference, the value in the Exit reference has precedence and replaces the corresponding value from the Exit object.

Return XML Buffer

Program exits have a single return value, which is an XML buffer. The return XML buffer has the following format:

```
<eTExitReturn>
  <eTExitReturnCategory> </eTExitReturnCategory>
  <eTExitReturnNative> </eTExitReturnNative>
  <eTExitLogMsg> </eTExitLogMsg>
  <eTExitContinue> </eTExitContinue>
  <eTExitCustom> </eTExitCustom>
</eTExitReturn>
```

Program exits must return an XML buffer with this format.

More information:

[Universal Provisioning Connector Reference](#) (see page 335)

UPC Exits DLL Interface

The DLL interface is the same as for Common Exits.

More information:

[Common Exits DLL Interface](#) (see page 218)

UPC Exits SOAP Interface

The SOAP-deployed program exits must present an external interface.

More information:

[Common Exits SOAP Interface](#) (see page 218)

eTExitType

Exit types determine the circumstances under which an exit is called. One of the types of exits is entered for eTExitType in the input XML buffer passed to the program exit.

Note: In all cases, the name of the object being passed is sent. This is formatted in both DN and Common Name format.

Valid Values for eTExitType

eTExitType Values for Managed and Non-managed Mode Exits

The following values are valid for eTExitType for both non-managed and managed mode exits:

INVOCATION_ERROR

Called by the Exit Interface library if it encounters a failure executing a UPC program exit.

ADD_ACCOUNT

Called when a create account request is received. Unlike the Modify operation, the password is passed to the Create operation as part of the account information.

DELETE_ACCOUNT

Called by the DELETE request. The account name is the only account attribute available to this exit.

MODIFY_ACCOUNT

Called by UPC when a modify account is invoked. The account information that is being passed to the modify account request is also passed to this program. For non-managed mode exits, the only exclusion to this is the password attribute.

RENAME_ACCOUNT

Called by the MODIFYRDN request. The new account name and new distinguished name are added to the current account name in the XML input buffer.

eTExitType Values for Non-Managed Mode Exits

The following exit types are valid for non-managed mode exits:

CHANGE_ACCOUNT_PASSWORD

Special case of MODIFY called when the UPC receives a password change for the account. The account name and the password attributes are the only information available to the program exit.

ENABLE_ACCOUNT

Special case of MODIFY called when the enable attribute for the account changes. The account name is the only account attribute available to this exit.

DISABLE_ACCOUNT

Special case of MODIFY called when the disable attribute for the account changes. The account name is the only account attribute available to this exit.

REQUEST_PENDING

Called by the UPC agent plug-in when a modify, add, or delete request successfully is sent to a non-managed system administrator. The account's request status is set to pending.

REQUEST_COMPLETED

Called by the UPC SASP agent plug-in when a modify request successfully marked the account's request status from pending to completed.

eTExitType Values for Service Level Agreement Monitor Exits

The following exit types are invoked only by the SLA Monitor service. For these exits, the XML Account block of the Input XML buffer is replaced with an XML Directory block:

```
<eTUPODirectory>
  <eTDN> </eTDN>
  <eTName> </eTName>
</eTUPODirectory>
```

eTDN

Distinguished name of the Directory.

eTName

Name of the Directory.

The SLA valid exit types are the following:

SLA_EXCEEDED

Called by the UPC Monitoring service when pending requests are found that exceed the SLA set for the corresponding directory. A list of accounts currently exceeding the SLA warning level is encoded in an Account block in the XML Directory Block.

SLA_WARNING

Called by the UPC Monitoring service when pending requests are found that exceed the SLA warning level set for the corresponding directory. A list of accounts currently exceeding the SLA warning level is encoded in an Account block in the XML Directory Block.

SLA_TIMEOFFSET

Called by the UPC Monitoring service before testing for accounts exceeding the SLA warning level. Used to skip non-business periods where the SLA may not apply. This exit must return an integer representing the number of seconds to subtract from the current time before determining whether a pending operation exceeds either of the two time limits.

eTExitType Values for Managed Mode Exits

The following exit types apply to managed mode exits:

READ_ACCOUNT

Called by the UPC Agent plug-in when a request to read an account is received. This exit must return the user data of the account in the eTExitCustom block of the XML return buffer.

Note: The format of the data returned by this exit is very important as policy objects need to use the same structure.

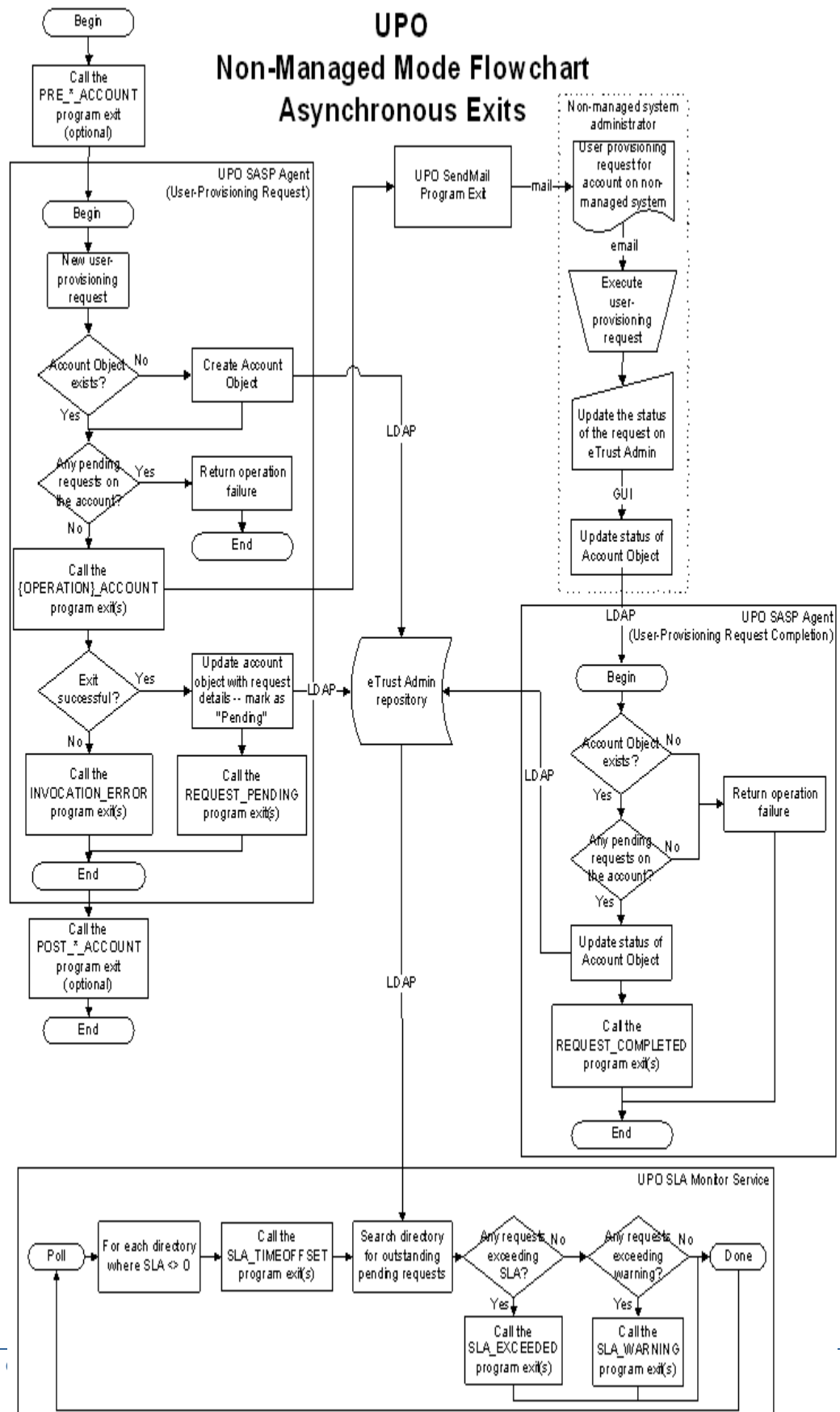
LIST_ACCOUNTS

Called by the UPC Agent plug-in when a request to enumerate accounts is received. This exit must return a list of account names.

Sample Flow/Execution Diagrams

Flowchart for Non-Managed Mode Exits

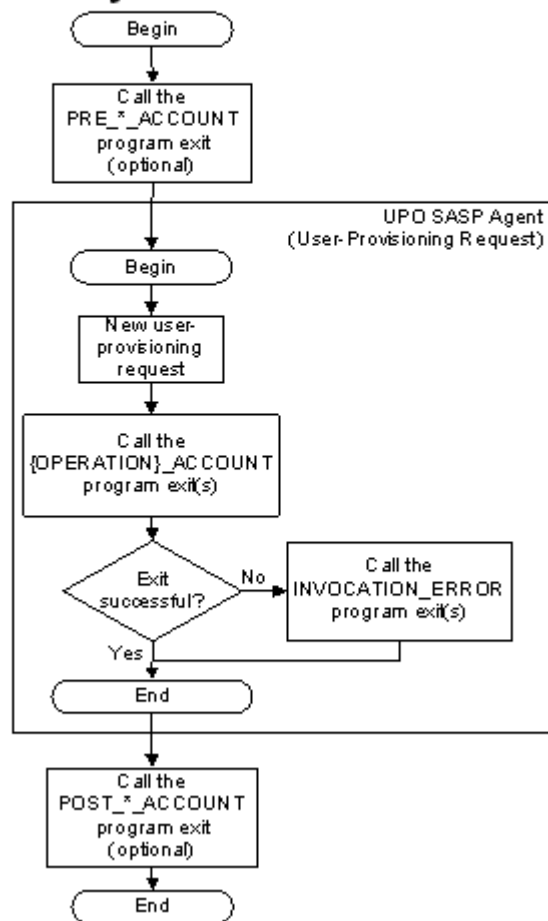
The following illustration provides a sample flowchart of the execution logic of asynchronous UPC program exits in non-managed mode:



Flowchart for Managed Mode Exits

The following diagram provides a sample flowchart of the execution logic of synchronous UPC program exits when in managed mode.

UPO Managed Mode Flowchart Synchronous Exits



Code Examples

Use the code examples located in the following directories as a guide when coding your UPC Program Exits:

Samples/UPO-SDK/UPOProgramExits
Samples/UPO-SDK/UPOSyncProgramExits

For an example of a SOAP exit (non-UPC specific), see the following:

Samples/ProgramExitSOAP

For an example of a Common Exit interfacing with UPC, see the following:

Samples/UPO-SDK/UPOCommonExits

Non-Managed Mode Exit Examples

A number of asynchronous exit examples are available in Samples/UPO-SDK/UPOProgramExits, as detailed below:

UpoSampleExit.cpp

- **UpoSampleExit**—Example of an exit which prints the XML input buffer into a log file in a readable format. This exit is usable with any exit type, including synchronous ones.
- **UpoAlwaysFail**—Simple exit that always returns a failure. Usable with any exit type.
- **UpoAlwaysFailButContinue**—Exit that always returns a failure but with the continue flag set to true. Usable with any exit type.
- **SlaSkipWeekends**—Example of a SLA_TIMEOFFSET exit to be used with the SLA Monitor service.

UpoGlobalUserExit.cpp

GetGlobalUser: Exit that finds the associated global user from a UPC account. Usable with any exit type, however the code sample currently checks for one of ADD_ACCOUNT, MODIFY_ACCOUNT or DELETE_ACCOUNT types. This example could also be used with synchronous exits.

UpoMarkCompletedExit.cpp

UpoMarkCompleted: Exit that marks the pending status of a UPC account to "completed". Intended to be used as a REQUEST_PENDING exit type. Note that using this exit effectively disables calls to the REQUEST_COMPLETED exit; if the latter exit needs to be called, use the Common Exit implementation instead.

UpoExitXMLBlock.cpp

Supporting C++ Class

Managed Mode Exit Examples

A synchronous mode exit example is available in the Samples/UPO-SDK/UPOSyncProgramExits folder.

The file names for the exit implementation and the methods implemented within them are as follows:

UpoSyncExit.cpp

AddAccount
DeleteAccount
ModifyAccount
ReadAccount
ListAccounts

Target.cpp

Modified C++ class from the SDK Connector example.

UpoExitXMLBlock.cpp

Supporting C++ Class

Common Exit with UPC Example

An example of a common exit method to be used as a POST exit to mark a UPC account as "completed" after a successful operation. The example is available in the Samples/UPO-SDK/UPOCommonProgramExits folder.

CommonExit.cpp**UpoMarkCompleted**

A common exit implementation, intended to be called from any POST account exit. This version will trigger a REQUEST_COMPLETED exit. The authentication mode should be PROXY_USER, configured with the etaupoad credentials.

Chapter 14: Supporting Program Exits In Connectors

This chapter covers information about supporting program exits in the connectors you have created.

This section contains the following topics:

[Execution Flow \(Logic\)](#) (see page 255)

[Support for Common Exits](#) (see page 257)

[Support for Native Exits](#) (see page 257)

[Exit Types](#) (see page 261)

[Code Examples for Program Exits in Options](#) (see page 262)

Execution Flow (Logic)

Program exits are referred to as either pre-exits or post-exits, depending on the operation that the Provisioning Server performs.

Pre-Exits

The Provisioning Server framework invokes a pre-exit before executing a particular operation. For common exits, the Provisioning Server framework interprets the return XML buffer to determine whether to continue execution of the particular operation. For native exits, the agent plug-in must interpret the return XML buffer.

If the agent plug-in returns a success status code (LDAP_SUCCESS), the Provisioning Server continues to perform the operation. If the agent plug-in returns an error code, the operation is aborted.

Note: For native program exits, your custom connector agent plug-in must interpret the return XML buffer. Furthermore, your agent plug-in must return either success or failure to the Provisioning Server framework. Success lets the operation continue. Failure aborts the operation.

If one pre-exit fails, the operation will be aborted even if other pre-exits let the operation continue. In addition, as soon as a pre-exit aborts the operation, other pre-exits (with the same or lower priority) will not be called.

Priority of Pre-Exits

Pre-exits are called in order of hierarchy and priority. If two program exits are referenced with the same priority, the order in which they are called is undefined. Priority guarantees only that higher priority exits are called before lower priority exits. The highest priority is "1" and the larger the number, the lower its priority.

More information:

[Program Exit Hierarchy and Order](#) (see page 212)

Operation

Once the Provisioning Server framework has invoked all pre-exits associated with the operation and each pre-exit lets the operation continue, the Provisioning Server framework executes the operation. Execution of the operation may result in another request to your agent plug-in.

Note: The agent plug-in receives multiple requests: one for the pre-exit, one for the operation, and one for the post-exit.

Post-Exit

Once the operation is completed successfully, the Provisioning Server framework invokes the post-exits referenced for that operation. Post-exits should be handled the same way as pre-exits. Your agent plug-in should not differentiate between a pre-exit and a post-exit, and you can use the exact same code to handle pre- and post-exits.

Priority of Post-Exits

Post-exits are called in order of hierarchy and priority. If two program exits are referenced with the same priority, the order in which the exits are called is not guaranteed. Priority only guarantees that higher priority exits are called before lower priority exits. The highest priority is "1" and the larger the number, the lower its priority.

More information:

[Program Exit Hierarchy and Order](#) (see page 212)

Support for Common Exits

Common exits are processed by the Provisioning Server framework. Thus, supporting common exits requires minimal changes. You only need to enhance your GUI plug-in. No agent plug-in change is needed.

Parser Table Enhancement

The parser table files already define new attributes for exit reference. Option parser table files include these parser table files, so you do not need to make any changes to your parser table.

GUI Plug-In Enhancement

Manager provides a standard property page for referencing program exits. To support common exits, you only need to add this property page to your account and endpoint property sheets. This enables your account and endpoint objects to reference program exits.

The standard exit reference property page is managed by the `CosExitRefPage` class in the COS module. For example, to add the property page to your account and directory property sheets, add the following line to your property sheet code:

```
propertyPages->AddTail(new CosExitRefPage(this));
```

Agent Plug-In Enhancement

No agent plug-in change is needed to support common exits.

Support for Native Exits

Native exits are processed by the endpoint. Thus, to support native exits, the endpoint must enhance both its GUI plug-in and agent plug-in.

Parser Table Enhancement

The parser table files already define new attributes for exit reference. Parser table files include these parser table files, so you do not need to make any changes to your parser table.

Since you are providing native program exits, you need to define your custom program exit object in the parser table. The exit object must have the following CLASS definition line:

```
CLASS Exit.<exit class name>,eTExit.<exit class name>,etavlcpr,secobjar
```

For example, the SDK defines the exit object class as follows:

```
CLASS Exit.SDKExit,eTExit.eTSDKExit,etavlcpr,secobjar
```

For a complete example of how to define an exit object, see the SDK `sdkparse.pty` sample file, which is provided with the SDK.

GUI Plug-In Enhancement

Exit Reference

The Provisioning Server GUI framework provides a standard property page for referencing program exits. To support common exits, you need to add the property page to your account and directory property sheets. This lets your account and endpoint objects reference program exits.

The standard exit reference property page is managed by the `CosExitRefPage` class in the COS module.

For example, to add `CosExitRefPage` to your account and directory property sheets, add the following line to your property sheet code:

```
propertyPages->AddTail(new CosExitRefPage(this));
```

Exit Definition

A property sheet must be provided to define a native program exit. This endpoint-specific exit definition property sheet is used to enter specific information that will be passed to the agent plug-in during exit invocation.

Your endpoint must define an XML format for this data, which is stored as an XML buffer in the `eTExitPayload` attribute in the custom program exit object. When an exit is invoked, the Provisioning Server framework sends this data to the agent plug-in. The agent plug-in parses the `eTExitPayload` attribute to get the data it needs to invoke the program exit.

Agent Plug-in Enhancement

Program Exit Invocation Request

The Provisioning Server framework sends all native program exit invocation requests to the endpoint. Even if the exit is referenced by an account object, the invocation request is sent to the code that manages the endpoint modify operation. Specifically, the directory `DEmodify()` function is called.

On a program exit invocation request, the Provisioning Server framework includes the `eTExitPayload` and `eTExitInvoke` attributes.

`eTExitPayload` contains the data regarding the definition of the exit. The value of `eTExitPayload` is the XML buffer stored in the program exit object that was defined by the program exit definition property sheet that you added to your GUI plug-in.

`eTExitInvoke` is an XML buffer. This data should be passed to the program exit, which needs to process it. The agent plug-in can process this information; however, often it does not need to do so.

The agent plug-in must be enhanced for performing the following tasks to support the program exit:

1. Determine whether an operation is an exit invocation request.
2. Invoke the program exit.
3. Interpret the result from the program exit.

More information:

[Program Exit Input Argument](#) (see page 213)

Determine Exit Invocation Request

Typically, the directory `DEmodify()` function processes requests to change values in the directory object. To support native program exits, you must enhance the directory `DEmodify()` function to also handle native exit invocation.

For a program exit invocation request, the Provisioning Server framework sends the `eTExitInvoke` attribute as part of the modify operation. The presence of the `eTExitInvoke` attribute is an indication that the request is an exit invocation request and not a normal modify request, as shown in the following example:

```
/*
|| The special attribute UTFEXITINVOKE indicates a request to invoke a
|| program exit.
*/
if (pMods->find_mod(UTFEXITINVOKE)) {

// Invoke the exit.

}
else {

// Normal directory modify request.

}
```

Invoke the Program Exits

You must define how your custom connector agent plug-in invokes the program exit.

The common exit has the following invocation methods:

- Through the DLL function call
- Through the SOAP method invocation

Your agent plug-in will probably define some other form of program exit invocation. That data is passed to the agent plug-in in the eTExitPayload attribute, which is an XML buffer.

The method of invoking program exits is to execute a command line utility. Thus the only information it needs is the utility name (including the path). You can define the SDK exit object as having a payload that only contains the full path to the utility.

The sample SDK exit payload is the following:

```
<eTSDKExit>
  <Program>program to execute</Program>
</eTSDKExit>
```

Interpret the Result from the Program Exit

Each program exit must return an XML buffer.

The agent plug-in must interpret this return XML buffer and return an appropriate status code to the Provisioning Server framework. For pre-exits, returning a success status to the Provisioning Server framework lets the operation continue. Returning a failure status will abort the operation.

Note: If the operation has multiple pre-exits, one pre-exit might return a success, which would let the operation continue; but another pre-exit could return failure, thus aborting the operation. If one pre-exit aborts the operation, it is aborted, even if other pre-exits let the operation continue. In addition, as soon as a pre-exit aborts the operation, other pre-exits with the same or lower priority will not be called.

More information:

[Program Exit Return Value](#) (see page 215)

Exit Types

The Provisioning Server framework only sends an exit invocation request to the agent plug-in if the exit type is one that can be handled by the agent plug-in. In general, your agent plug-in does not need to handle exit types. However, if your custom connector only permits certain program exit types, it must check the eExitType tag attribute in the eExitInvoke attribute.

Exit Type Functionality

Exit type is a value that determines the circumstances under which an exit is called. One of the types of exits is entered for eExitType (in the input XML buffer passed to the program exit). The first 12 exit types (values) can be common or native exits, that is, common code and namespace (connector) code can be triggered to process them. The remaining exit types, however, can be common exits only. It should be noted that not all program exit types are referenced from various object types.

Notes: In all cases the name of the object being passed is sent. This is formatted in both DN and Common Name format.

To have complete control over passwords (either at the global user or the account levels), you must provide exits both for create user/account and for change password user/account. In other words, for new global users (accounts), the change password exit is not called. For new global users (accounts), the password is passed in as part of the attribute for the create exit (for example, PRE_CREATE_GLOBAL_USER).

More information:

[Valid Values for Exit Types](#) (see page 221)

Code Examples for Program Exits in Options

See the SDK sample connector. The exit handling code is in the SDKDirectory.cpp file.

The method SDKDirectory::DEmodify() determines whether a request is an exit invocation request, and if it is, calls the SDKDirectory::InvokeExit() method.

Appendix A: Provisioning Schema and Structure

The provisioning schema and structure are required when you perform one of the following:

- Use Batch Utility or any other general purpose LDAP utility to construct batch processes interfacing with the Provisioning Server.
- Build, interpret, or modify LDAP Interface File Format (LDIF) files to work with Provisioning Server data and combine it with data from other LDAP-enabled applications.

For more information, see the *CA Identity Manager Provisioning Guide* and the endpoint-specific connector guides.

This section contains the following topics:

[Provisioning Schema](#) (see page 263)

[Endpoint Structure](#) (see page 266)

Provisioning Schema

A *provisioning schema* consists of the object classes, attributes in object classes, and attribute types.

All of this information is necessary when constructing syntactically correct LDAP operations, such as SEARCH, ADD, MODIFY, and DELETE.

COSSchemaAbridged.txt File

The abridged version of common objects provisioning schema can be generated using the following command:

```
dumpptt -t cosparse> COSSchemaAbridged.txt
```

where the output is stored in COSSchemaAbridged.txt.

This file provides a list of every object class and attribute in the common objects schema. For each attribute, only the most commonly used keywords are supplied. Use this file if you are constructing LDAP-compatible files for any of the batch processes.

COSSchemaUnabridged.txt File

The full definition of common objects provisioning schema can be generated using the following command:

```
dumpptt -t cosparse -f > COSSchemaFull.txt
```

where the output is redirected to COSSchemaFull.txt.

This file provides a complete list of each object class and attribute in the common objects schema. It includes all the information provided in the COSSchemaAbridged.txt file, as well as additional information required when parsing, formatting, and presenting the data received from the common objects. Use this file if you need more detailed information for the object classes and attributes in the common objects.

File Formats

The format of these files is defined using the following distinct definitions:

- Object Class definitions
- Attribute definitions

Object Class Definitions

The lines that define the object classes are in the following form:

```
CLASS user_friendly_name
  LDAP ObjectClass Name : ldap_name
  ExternalName: external_name
  NamingAttributes: naming_attribute
```

user_friendly_name

Specifies the user-friendly object class name.

ldap_name

Specifies the LDAP name used for defining the schema.

external_name

Specifies the relative distinguished name (RDN) value for containers.

naming_attribute

Specifies the RDN attribute.

Attribute Definitions

Directly beneath the object class definition are several attribute lines. These lines define the attribute types in the object class.

```
ATTRIBUTE (LDAP Name) ldap_object_class_name::ldap_attribute_name
    User-friendly Name: user_friendly_name
    Description: Global description
    ProhibitedCharacters: characters
    MinLength: integer
    MaxLength: integer
    EditType: data_type
    IsSpaceAllowedIn: boolean
    IsAsciiOnly: boolean
    IsMultiValued: boolean
    Case: sensitivity
```

ldap_object_class_name: :ldap_attribute_name

Designates the LDAP name used for the object class and the LDAP name of the attribute.

User-friendly Name: *user_friendly_name*

Designates the user-friendly name.

Description: Global *description*

Describes of the attribute.

ProhibitedCharacters: *characters*

Specifies a string of characters prohibited in the attribute.

MinLength: *integer*

Specifies the minimum length of the attribute value.

MaxLength: *integer*

Specifies the maximum length of the attribute value.

EditType: *data_type*

Specifies the type of data for this LDAP attribute.

IsSpaceAllowedIn: *Boolean*

Boolean value that identifies whether spaces are permitted.

IsAsciiOnly: *Boolean*

Boolean value that determines whether the attribute supports ASCII values.

IsMultiValued: *Boolean*

Boolean value that determines if the attribute is multi-valued.

Case: *sensitivity*

String that identifies whether the attribute can contain uppercase or lowercase characters. Valid values for *sensitivity* are insensitive, insensitive-upper, insensitive-lower, sensitive, sensitive-upper, or sensitive-lower.

Endpoint Structure

The hierarchical relationship that exists between the objects in the endpoint is important to the endpoint schema. This relationship is expressed through a endpoint structure called the Data Information Tree (DIT). Knowing the hierarchy is essential to constructing syntactically correct endpoint operations.

Distinguished Names

Distinguished names (DNs) identify the objects in an endpoint. DNs contain a sequence of individual entries that specifies the location of an object in the DIT. A DN is similar to a file system path name.

In the Provisioning Server, the format of the DN consists of two parts, a base DN and a domain name suffix. The base DN specifies the DN of an object without any domain information. You must specify only the base DN when writing batch processes.

For example, a base DN of a global user object is as follows:

```
eTGlobalUserName=global_user_name,  
eTGlobalUserContainerName=Global Users,  
eTNamespaceName=CommonObjects
```

The domain name suffix is the combination of the provisioning domain suffix and the Provisioning Server suffix (dc=eta). You must specify the domain name suffix and the base DN when writing LDIF files.

For example, if your provisioning domain name is usa, the domain name suffix for your domain is:

```
dc=usa,dc=eta
```

When accessing a global user using an LDIF file, the DN would look like the following:

```
eTGlobalUserName=global_user_name,
eTGlobalUserContainerName=Global Users,
eTNamespaceName=CommonObjects,
dc=usa,
dc=eta
```

Common Object DITs

The Provisioning Server provides the following DITS for common objects:

- A DIT containing common objects that are stored in the Provisioning Server top-level root domain
- A DIT containing common objects that are stored in the provisioning domain.

Provisioning Server Root Domain DIT

The following table contains the common objects that are stored in the Provisioning Server top-level root domain. This DIT is stored only in the root domain but is used for the provisioning domain.

LDAP Object Name	DN of Object Instance
eTNamespace	eTNamespaceName=CommonObjects
eTDSAContainer	eTDSAContainerName=DSAs, eTNamespaceName=CommonObjects
eTDSA	eTDSAName= <i>dsa_name</i> , eTDSAContainerName=DSAs, eTNamespaceName=CommonObjects
eTRequestContainer	eTRequestContainerName=Requests, eTNamespaceName=CommonObjects
eTPasswordResetRequest	eTPasswordResetID= <i>password_reset_id</i> , eTRequestContainerName=Requests, eTNamespaceName=CommonObjects
eTSystemSettingsContainer	eTSystemSettingsContainerName=System Settings, eTNamespaceName=CommonObjects
eTSystemSettings	eTSystemSettingsName=Settings, eTSystemSettingsContainerName=System Settings, eTNamespaceName=CommonObjects

Provisioning Domain DIT

The following table contains the common objects that are stored in the provisioning domain:

LDAP Object Name	DN of Object Instance
eTNamespace	eTNamespaceName=CommonObjects
eTAdminProfileContainer	eTAdminProfileContainerName=Admin Profiles, eTNamespaceName=CommonObjects
eTAdminProfile	eTAdminProfileName= <i>admin_profile_name</i> , eTAdminProfileContainerName=Admin Profiles, eTNamespaceName=CommonObjects
eTConfigContainer	eTConfigContainerName=Configuration, eTNamespaceName=CommonObjects
eTConfig	eTConfigName= <i>config_name</i> , eTConfigContainerName=Configuration, eTNamespaceName=CommonObjects
eTDSAContainer	eTDSAContainerName=DSAs, eTNamespaceName=CommonObjects
eTDSA	eTDSAName= <i>dsa_name</i> , eTDSAContainerName=DSAs, eTNamespaceName=CommonObjects
eTExitContainer	eTExitContainerName=Program Exits, eTNamespaceName=CommonObjects
eTExit	eTExitName= <i>exit_name</i> , eTExitContainerName=Program Exits, eTNamespaceName=CommonObjects
eTGlobalGroupContainer	eTGlobalGroupContainerName= Global User Groups, eTNamespaceName=CommonObjects
eTGlobalGroup	eTGlobalGroupName= <i>global_group_name</i> , eTGlobalGroupContainerName=Global User Groups, eTNamespaceName=CommonObjects
eTGlobalUserContainer	eTGlobalUserContainerName=Global Users, eTNamespaceName=CommonObjects

LDAP Object Name	DN of Object Instance
eTGlobalUser	eTGlobalUserName= <i>global_user_name</i> , eTGlobalUserContainerName=Global Users, eTNamespaceName=CommonObjects
eTInclusionContainer	eTInclusionContainerName=Inclusions, eTNamespaceName=CommonObjects
eTInclusionSuperior	eTSuperiorClass= <i>ldap_class_name</i> , eTInclusionContainerName=Inclusions, eTNamespaceName=CommonObjects
eTInclusionSubordinate	eTSubordinateClass= <i>ldap_class_name</i> , eTSuperiorClass= <i>ldap_class_name</i> , eTInclusionContainerName=Inclusions, eTNamespaceName=CommonObjects
eTInclusionObject	eTInclusionID= <i>inclusion_id</i> , eTSubordinateClass= <i>ldap_class_name</i> , eTSuperiorClass= <i>ldap_class_name</i> , eTInclusionContainerName=Inclusions, eTNamespaceName=CommonObjects
eTOperationContainer	eTOperationContainerName=Operations, eTNamespaceName=CommonObjects
eTOperation	eTOperationID= <i>operation_id</i> , eTOperationContainerName=Operations, eTNamespaceName=CommonObjects
eTPasswordProfileContainer	eTPasswordProfileContainerName=Password Profile, eTNamespaceName=CommonObjects
eTPasswordProfile	eTPasswordProfileName= <i>password_profile_name</i> , eTPasswordProfileContainerName=Password Profile, eTNamespaceName=CommonObjects
eTRoleContainer	eTRoleContainerName=Roles, eTNamespaceName=CommonObjects
eTRole	eTRoleName= <i>role_name</i> , eTRoleContainerName=Roles, eTNamespaceName=CommonObjects
eTXXXPolicyContainer	eTXXXPolicyContainerName= <i>namespace</i> Policies, eTNamespaceName=CommonObjects

LDAP Object Name	DN of Object Instance
eTXXXPOLICY	eTXXXPOLICYName= <i>policy_name</i> , eTXXXPOLICYContainerName= <i>namespace</i> Policies, eTNamespaceName=CommonObjects

User-Friendly Object Names

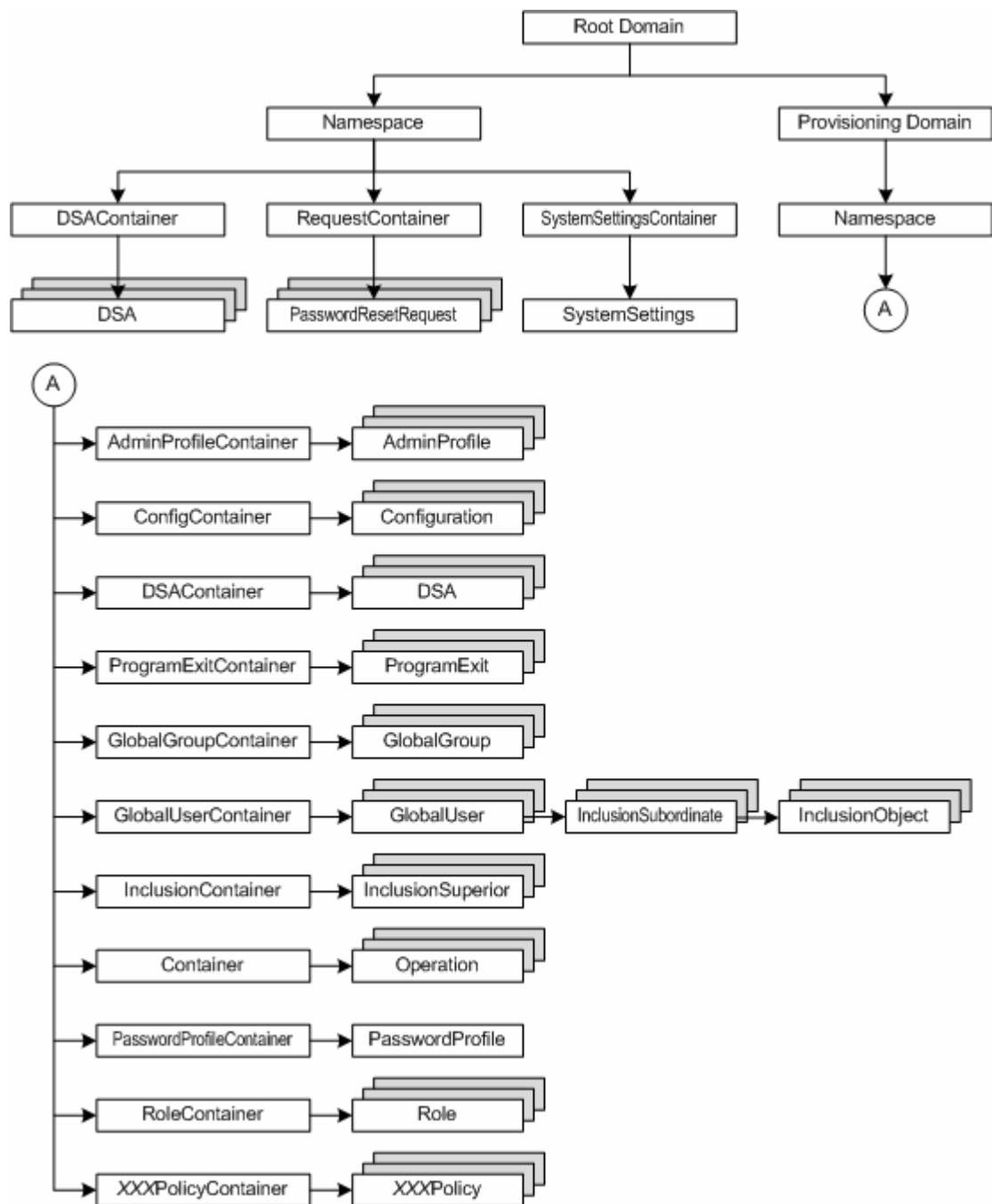
The following table lists alphabetically the LDAP object names and their user-friendly names:

LDAP Object Name	User-Friendly Name	Description
eTAdminProfile	AdminProfile	Admin profile name
eTAdminProfileContainer	AdminProfileContainer	Admin profiles container
eTConfig	Config	Configuration object
eTConfigContainer	ConfigContainer	Configuration object container
eTDSA	DSA	DSA name
eTDSAContainer	DSAContainer	DSAs container
eTExit	ProgramExit	Program exit
eTExitContainer	ProgramExitContainer	Program exit container
eTGlobalGroup	GlobalGroup	Global user group name
eTGlobalGroupContainer	GlobalGroupContainer	Global user groups container
eTGlobalUser	GlobalUser	Global user name
eTGlobalUserContainer	GlobalUserContainer	Global users container
eTInclusionContainer	InclusionContainer	Inclusions container
eTInclusionObject	InclusionObject	Inclusion object
eTInclusionSubordinate	InclusionSubordinate	Subordinate class inclusion
eTInclusionSuperior	InclusionSuperior	Superior class inclusion
eTNamespace	Namespace (connector)	Namespace name
eTOperation	Operation	Operation ID
eTOperationContainer	OperationContainer	Operations container

LDAP Object Name	User-Friendly Name	Description
eTPasswordProfile	PasswordProfile	Password profile
eTPasswordProfileContainer	PasswordProfileContainer	Password profile container
eTPasswordResetRequest	PasswordResetRequest	Password reset request
eTRequestContainer	RequestContainer	Requests container
eTRole	Role	Role name
eTRoleContainer	RoleContainer	Roles container
eTSystemSettings	SystemSettings	System settings
eTSystemSettingsContainer	SystemSettingsContainer	System settings container
eTXXXPolicy	XXXPolicy	Policy name
eTXXXPolicyContainer	XXXPolicyContainer	Policies container

Object Hierarchy

The following illustration shows the hierarchy of the entries in the common objects.



Appendix B: GUI Framework API Functions

This section contains the following topics:

[apply\(\)](#) (see page 273)

[setData\(\)](#) (see page 275)

[setErrorField\(\)](#) (see page 277)

apply()

Purpose

The `apply( )` function updates any attributes on a property page that have changed after a user clicks the OK, Apply, or Enter buttons.

Syntax

```
virtual int apply( )
```

Arguments

None.

Return Value

This function returns one of the following values:

- `PAGE_MODIFIED` if data has changed
- `PAGE_UNMODIFIED` if no data has changed
- `PAGE_ERROR` if any errors are detected

Comments

The GUI framework provides a set of overloaded functions called `setSdsData()`.

For the following MFC controls and objects, these functions update the attributes on a property page directly from an MFC control or object. You can call this routine in one of the following ways:

```
BOOL setSdsData (PWCHAR_T pszProperty,
                 CEdit *ctb,
                 BOOL isNumeric = FALSE,
                 BOOL forceUpdate = FALSE);

BOOL setSdsData (PWCHAR_T pszProperty,
                 CButton * cb,
                 BOOL forceUpdate = FALSE);

BOOL setSdsData (PWCHAR_T pszProperty,
                 CSpinButtonCtrl * sb,
                 BOOL forceUpdate = FALSE);

BOOL setSdsData (PWCHAR_T pszProperty,
                 CComboBox * cb,
                 BOOL forceUpdate = FALSE);

BOOL setSdsData (PWCHAR_T pszProperty,
                 CDateTimeCtrl * db,
                 BOOL forceUpdate = FALSE);

BOOL setSdsData (PWCHAR_T pszProperty,
                 CStringArray * psa,
                 BOOL forceUpdate = FALSE);
```

pszProperty is the name of the property. The second argument in these functions is a pointer to the corresponding MFC control or object. Set *forceUpdate* to force the value to be updated, even when the value is not changed. For the CEdit control, *isNumeric* forces the value to be treated as a number.

The `setSdsData( )` functions return TRUE if the value of the control has changed. Unless an error has occurred on the page, if any call to `setSdsData( )` returns TRUE, you should set the return value of the `apply( )` function to `PAGE_MODIFIED`.

For the following MFC controls, you must manually retrieve the new value and compare it to the original value. If the value has changed, then the change must be sent back to the GUI framework. This can be done using one of the following calls, depending on whether the value is a string or an integer:

```
void setSdsData (PWCHAR_T pszProperty, const PWCHAR_T pszData);
void setSdsData (PWCHAR_T pszProperty, int iData);
```

Note: This routine must validate all data and display an appropriate message to the user if an error occurs. Unlike the `setSdsData( )` functions in the first group, these functions do not provide error checking.

Note: The `apply( )` function is declared in the `CAPropertyPage.h` file.

setData()

Purpose

The `setData( )` function initializes the property page when it appears. It retrieves the attribute values from the DIT and initializes the corresponding MFC controls or objects.

Syntax

```
virtual void setData( )
```

Arguments

None.

Return Value

None.

Comments

The CAPropertyPage base class provides access to the SDS buffer containing the initial values of all attributes and their values. The SDS buffer can be accessed through the propertySheet->sdsOriginal pointer.

You must initialize the value of all fields on your property page in this routine.

For certain MFC controls (CEdit, CButton, CSpinButtonCtrl, CComboBox, and CDateTimeCtrl) and objects (CStringArray), you can use the setData() overload function to set the MFC control from the SDS buffer.

When the setData() function is used to initialize the standard controls, use one of the following signatures:

```
void setData (PWCHAR_T pszProperty, CEdit * ctb, BOOL isNumeric = FALSE);
void setData (PWCHAR_T pszProperty, CButton * cb);
void setData (PWCHAR_T pszProperty, CSpinButtonCtrl * sb);
void setData (PWCHAR_T pszProperty, CComboBox * cb);
void setData (PWCHAR_T pszProperty, CDateTimeCtrl * db);
void setData (PWCHAR_T pszProperty, CStringArray * psa);
```

pszProperty, is a pointer to the property name. The second argument is a pointer to the MFC control or object. The setData() call initializes the MFC control or object from the corresponding property value in the SDS buffer.

When using other MFC controls, you must manually set their values, retrieving the data from the SDS buffer and using it to initialize the control. The calls are as follows, depending on whether it is a string or an integer:

```
PWCHAR_T string_value = propertySheet->sdsOriginal-> getString (PWCHAR_T
prop_name)

int CASdsBuffer::getInt (PWCHAR_T prop_name)
```

Note: If necessary, you may need to convert these values to the proper form and use them to initialize the MFC control.

The CAPropertyPage base class provides a Boolean variable called bInit. This variable is for initialization code that is used only once, and therefore, initialized only once. This variable is set to FALSE the first time a property page appears; after the first initialization, the flag is set to TRUE.

Possible uses for this flag are as follows:

- For code that might be placed in the MFC routine OnInitDialog(), you can add the (!bInit) clause instead.
- For SubClassDlgItem() calls that are used to link the data elements with their corresponding MFC controls, you can add the calls to the (!bInit) clause.

Note: You could link the data elements with their corresponding MFC controls by using the MFC DoDataExchange() family of functions. In this case, you do not need the SubClassDlgItem() calls.

Note: The setData() function is declared in the CAPropertyPage.h file.

setErrorField()

Purpose

The setErrorField() function sets an error flag when an error is encountered, for example, when a user types a character that is not valid for an account.

Syntax

```
virtual CWnd * setErrorField (PWCHAR_T errorField)
```

Arguments

errorField

Name of the field in error. The value should be the property name.

Return Value

Address of the field in error. This address should be a pointer to an MFC control.

Comments

When the name of a property is passed to this function, it must return the address of the control corresponding to that property.

Note: The setErrorField() function is declared in the CAPropertyPage.h file.

Appendix C: GUI Callout Functions

A GUI callout module contains the entry points to your GUI plug-in. By using these entry points, your GUI plug-in can use GUI features, such as dragging and dropping objects. This appendix is an alphabetical reference to all the GUI callout functions in the GUI framework.

The GUI callout functions are defined in `etoload.h` and are C-style interfaces.

Note: Many of these functions do not need to be implemented in your plug-in. Most have defaults that are appropriate for most situations.

This section contains the following topics:

[CanDuplicateEntry](#) (see page 279)
[CreateInclusion](#) (see page 280)
[DeleteEntry](#) (see page 282)
[DuplicateEntry](#) (see page 283)
[GetPropertySheetID](#) (see page 284)
[IsValidDrag](#) (see page 286)
[OnCommand](#) (see page 287)
[RenameEntry](#) (see page 288)

CanDuplicateEntry

Purpose

The `CanDuplicateEntry` function determines whether an object can be duplicated. This function is used with the `DuplicateEntry` function.

If this function is not defined in the GUI plug-in, the default action is to return 1, indicating that duplication is permitted.

Syntax

```
int CanDuplicateEntry(  
    HWND      hGui,  
    HANDLE    hptt,  
    ETA_HANDLE hRepository,  
    PWCHAR_T  pszClass,  
    PWCHAR_T  pszSourceDN);
```

Arguments***hGui***

Handle to a Manager window.

hptt

Handle to the Provisioning Server parser API.

hRepository

Handle for the Provisioning Server connection.

pszClass

Object class of the selected object.

pszName

DN of the selected object.

Return Value

The value 0 (FALSE) specifies that duplicating is not possible, while the value 1 (TRUE) specifies that duplication is permitted.

CreateInclusion

Purpose

The CreateInclusion function forms an inclusion between two objects.

If this function is not defined in the GUI plug-in, the default action is to create a standard inclusion..

Syntax

```
int CreateInclusion(  
    CWND      hGui,  
    HANDLE    hptt,  
    ETA_HANDLE hRepository,  
    PWCHAR_T  pszParentClass,  
    PWCHAR_T  pszParentDN,  
    PWCHAR_T  pszChildClass,  
    PWCHAR_T  pszChildDN,  
    PWCHAR_T  pszRelationship,  
    void **    ppSds);
```


Arguments***hGui***

Handle to a Manager window.

hptt

Handle to the Provisioning Server parser API.

hRepository

Handle for the Provisioning Server connection.

pzParentClass

Object class of the parent object.

pszParentDN

DN of the parent object.

pzChildClass

Object class of the child object.

pszChildDN

DN of the child object.

pszRelationship

Identifies the relationship (typically, this argument is empty).

ppSds

Address of a pointer to the buffer containing the results.

Return Values**ETA_GUIEXIT_PASSTHRU**

Indicates to the framework to perform the default action

ETA_*

Other return values from etaerror.h

DeleteEntry

Purpose

The DeleteEntry function deletes an object.

If this function is not defined in the GUI plug-in, the default action is to generically delete the object.

Syntax

```
int DeleteEntry (
    ETA_HANDLE hRepository,
    HANDLE      hptt,
    PWCHAR_T    pszClass,
    PWCHAR_T    pszName,
    void **      ppSds );
```

Arguments

hRepository

Handle for the Provisioning Server connection.

hptt

Handle to the Provisioning Server parser API.

pszClass

Object class of the selected object.

pszName

DN of the selected object.

ppSds

Address of a pointer to the buffer containing the results.

Return Values

ETA_GUIEXIT_PASSTHRU

Indicates to the framework to perform the default action

ETA_*

Other return values from etaerror.h

DuplicateEntry

Purpose

The DuplicateEntry function is used with the CanDuplicationEntry function. It is called when duplicating an object.

If this function is not defined in the GUI plug-in, the default action is to generically duplicate the object.

Syntax

```
int DuplicateEntry (
    HWND      hGui,
    HANDLE     hptt,
    ETA_HANDLE hRepository,
    PWCHAR_T   pszClass,
    PWCHAR_T   pszSourceDN,
    PWCHAR_T   pszTargetDN,
    BOOL_T     withInclusions,
    void **     ppSds );
```

Arguments

hGui

Handle to a Manager window.

hptt

Handle to the Provisioning Server parser API.

hRepository

Handle for the Provisioning Server connection.

pszClass

Object class of the selected object.

pszSourceName

DN of the selected object.

pszTargetName

DN of the target object.

withInclusions

Indicates whether inclusions should also be duplicated.

ppSds

Address of a pointer to the buffer containing the results.

Return Values

ETA_GUIEXIT_PASSTHRU

Indicates to the framework to perform the default action

ETA_*

Other return values from etaerror.h

GetPropertySheetID

Purpose

The GetPropertySheetID function returns a pointer to a newly allocated property sheet. The C++ new operator is expected to be used.

If this function is not defined in the GUI plug-in, the default action is to return a NULL pointer.

Syntax

```
void * GetPropertySheetID (
    void          *pParentWnd,
    PWCHAR_T      pszClassName,
    PWCHAR_T      pszObjectDN,
    PWCHAR_T      pszObjectID,
    HWND          hGui,
    ETA_HANDLE     hRepository,
    HANDLE         hptt,
    BOOL_T         isNewObject,
    BOOL_T         isModal,
    BOOL_T         confirmDelete,
    BOOL_T         persistentSelects,
    BOOL_T         isDupObject );
```

Arguments****pParentWnd***

Pointer to a Manager window.

pszClassName

Object class name of the selected object.

pszObjectDN

DN of the selected object.

pszObjectID

ID of the selected object, which may be a NULL pointer.

hGui

Handle to a Manager window.

hRepository

Handle for the Provisioning Server connection.

hptt

Handle to the Provisioning Server parser API.

isNewObject

This is for a new object.

isModal

The sheet displays as a modal (pop-up) property sheet.

confirmDelete

A dialog appears confirming the delete or the delete is automatic. This parameter is passed on to the constructor of a CAPropertySheet derived class.

persistentSelects

A persistent selection criteria is used.

IsDupObject

This is for a duplicate object.

Return Value

Pointer to a valid property sheet.

IsValidDrag

Purpose

The IsValidDrag function enables a user to drag an object over another object.

If this function is not defined in the GUI plug-in, the default action is to permit the drag, as long as the drag and target directories have the same type and name (DN).

Syntax

```
int IsValidDrag (
    ETA_HANDLE hRepository,
    HANDLE      hptt,
    PWCHAR_T    pszDragClass,
    PWCHAR_T    pszDragObject,
    PWCHAR_T    pszTargetClass,
    PWCHAR_T    pszTargetObject,
    PWCHAR_T    pszDragDirectoryClass,
    PWCHAR_T    pszDragDirectoryName,
    PWCHAR_T    pszTargetDirectoryClass,
    PWCHAR_T    pszTargetDirectoryName );
```

Arguments

hRepository

Handle for the Provisioning Server connection.

hptt

Handle to the Provisioning Server parser API.

pszDragClass

Object class of the object that is being dragged.

pszDragObject

DN of the object that is being dragged.

pszTargetClass

Object class of the object that it is passing over.

pszTargetObject

DN of the object that it is passing over.

pszDragDirectoryClass

Directory class of the object that is being dragged.

pszDragDirectoryName

Directory name of the object that is being dragged.

pszTargetDirectoryClass

Directory class of the object that it is passing over.

pszTargetDirectoryName

Directory name of the object that it is passing over.

Return Value

The value 0 specifies that dragging is not supported, while the value 1 specifies that dragging is supported.

OnCommand

Purpose

The OnCommand function executes the selected menu command for an agent plug-in.

If this function is not defined in the GUI plug-in, the default action is to do nothing, and return FALSE.

Syntax

```
BOOL_T OnCommand (  
    ETA_HANDLE hRepository,  
    HANDLE     hptt,  
    PWCHAR_T   pszClass,  
    PWCHAR_T   pszDN,  
    UINT       commandId );
```

Arguments***hRepository***

The handle for the Provisioning Server connection.

hptt

The handle to the Provisioning Server parser API.

pszClass

The object class of the selected object.

pszDN

The DN of the selected object.

commandId

The ID of the command from the menu.

Return Value

The value TRUE specifies that the command was processed, while the value FALSE specifies that the command was not processed.

RenameEntry

Purpose

The RenameEntry function is used when renaming an object. If this function is called, the GUI plug-in must provide the dialog or property sheet and make changes to the SDS buffer directly.

If this function is not defined in the GUI plug-in, the default action is to try to generate a generic object rename operation.

Syntax

```
int RenameEntry(  
    HWND      hGui,  
    HANDLE     hptt,  
    ETA_HANDLE hRepository,  
    PWCHAR_T   pszClass,  
    PWCHAR_T   pszDN,  
    void **     ppSds );
```

Arguments

hGui

The handle to a Manager window.

hptt

The handle to the Provisioning Server parser API.

hRepository

The handle for the Provisioning Server connection.

pszClass

The object class of the selected object.

pszDN

The DN of the selected object.

ppSds

The address of a pointer to the buffer containing the results.

Return Values

ETA_GUIEXIT_PASSTHRU

Indicates to the framework to perform the default action

ETA_*

Other return values from etaerror.h

Appendix D: Superagent Framework APIs

The superagent framework APIs contain a set of functions for each object class of an agent plug-in.

This section contains the following topics:

[DEadd](#) (see page 289)
[DEdelete](#) (see page 290)
[DEinit](#) (see page 291)
[DEinitNewObject](#) (see page 292)
[DEmodify](#) (see page 293)
[DEmodrdrn](#) (see page 293)
[DEsearch](#) (see page 294)

DEadd

Purpose

The DEadd API performs an LDAP ADD request. This API is found in DMODirectoryEntry.H.

Syntax

```
virtual int DEadd (
                    DMOAddOp      *pAddOp );
```

Arguments

*pAddOp

Pointer to the operation object that has all the data for the object being added.

Return Value

LDAP error code. The value LDAP_SUCCESS represents a successful add; any other LDAP error code represents an unsuccessful add. If the object already exists, the code should return LDAP_ALREADY_EXISTS.

DDelete

Purpose

The DDelete API performs an LDAP DELETE request.

This API is found in DModirectoryEntry.H.

Syntax

```
virtual int DDelete (
    DModDeleteOp *pDeleteOp );
```

Arguments

***pDeleteOp**

Pointer to an operation object that has all data for the object being deleted.

Return Value

LDAP error code. The value LDAP_SUCCESS represents a successful deletion; any other LDAP error code represents an unsuccessful delete. If the object does not exist, the code should return LDAP_NO_SUCH_OBJECT.

Comments

The DModirectoryEntry class provides a detach() function that helps remove a node from the DIT. When an internal node is removed from the DIT, all of the node's children are removed. For leaf nodes, your agent plug-in must delete the account, group, or any other object from the connector server. Do not use the detach() function.

Example

```
int CSampleInternalNode::DDelete(DModDeleteOp *pDeleteOp)
{
    // Detach this node from the DIT.
    detach();

    // Agent plug-in specific code
    return 0;
}
int CSampleLeafNode::DDelete(DModDeleteOp *pDeleteOp)
{
    // Agent plug-in specific code: Remove the account from the target
    system.

    // Return error code.
    return rc;
}
```

DEinit

Description

The DEinit API initializes an object in each object class.

This API is found in DMODirectoryEntry.H.

Syntax

```
virtual int DEinit (
    DMO_LDAP_Entry      *pEntry,
    DMOMessage          *pStatusMsg );
```

Arguments

***pEntry**

Contains the values of required and optional attributes needed to initialize the object class. *pEntry* can be null if the object class does not require initialization values.

***pStatusMsg**

Pointer to an object to store the error message if the initialization is unsuccessful.

Return Value

LDAP error code. The value LDAP_SUCCESS represents a successful initialization; any other LDAP error code represents an unsuccessful initialization.

Comments

If you want your object class to be a logging component, register it as a logging component. To register your object class as a logging component, you must call the DMODirectoryEntry class method using registerLogComponent().

You must create or establish child nodes for each object class that can appear as a child of this class.

DEinitNewObject

Purpose

The DEinitNewObject initializes a new object when creating a new object in the target system.

This API is found in DMODirectoryEntry.H. Its default action is to call DEinit().

Syntax

```
virtual int DEinitNewObject (
    DMODirectoryEntry *pProposedParent,
    DMOAddOp *pAddOp );
```

Arguments

***pProposedParent**

Pointer to the parent object for the new object that you are adding.

***pAddOp**

Pointer to the operation object that has all the data for the object being added.

Return Value

LDAP error code. The value LDAP_SUCCESS represents a successful initialization; any other LDAP error code represents an unsuccessful initialization.

Comment

As part of its operation, the superagent may need to create a new object class. To indicate whether the create requests come from the superagent itself or from the client of the superagent, there are two initialization functions. The DEinitNewObject() function is used indicate that the create requests come from the client. In general, most object classes do not need to make this distinction. Thus, these object classes do not need to implement the DEinitNewObject() method. The DMODirectoryEntry class method DEinitNewMethod() simply calls the DEinit() method.

DModify

Purpose

The DModify API performs an LDAP MODIFY request.

This API is found in DModDirectoryEntry.H

Syntax

```
virtual int DModify (  
    DMOModifyOp  *pModifyOp );
```

Arguments

*pModifyOp

Pointer to the operation object that has all the data for the object that you are modifying.

Return Value

LDAP error code. The value LDAP_SUCCESS represents a successful modify; any other LDAP error code represents an unsuccessful modify.

DModrdn

Purpose

The DModrdn API performs an LDAP MODRDN, which renames the DN of an object.

This API is found in DModDirectoryEntry.H.

Syntax

```
virtual int DModrdn (  
    DMOModrdnOp  *pModrdnOp );
```

Arguments

*pModrdnOp

Pointer to the operation object that has all the data for the objecting being moved or renamed.

Return Value

LDAP error code. The value LDAP_SUCCESS represents a successful rename; any other LDAP error code represents an unsuccessful rename.

DEsearch

Purpose

The DEsearch API performs an LDAP SEARCH request. The default action is to support a simple BASE LEVEL search. It returns the DN and OBJECTCLASS of the searched node.

This API is found in DMODirectoryEntry.H.

Syntax

```
virtual int DEsearch (
    DMOSearchOp  *pSearchOp );
```

Arguments

*pSearchOp

Pointer to the operation object that has all the data for the object being searched for. This included the type of search that you want to perform. You can perform one of the following types of searches:

- SCOPE_BASE

Searches for the object specified in the base DN.

- SCOPE_ONELEVEL

Searches one level below the search base. For example, if you are searching for a global user in a specific department, perform a SCOPE_ONELEVEL search.

If the search scope value is less than or equal to 0, a SCOPE_ONELEVEL search is performed. If the search scope value is greater than 0, a SCOPE_BASE search is performed.

Return Value

LDAP error code. The value LDAP_SUCCESS represents a successful search; any other LDAP error code represents an unsuccessful search.

Comments

- Static internal nodes that do not have any attributes must use the default DEsearch() function. For these types of nodes, your agent plug-in does not have to implement a DEsearch() function.
- If the node has attributes and you want to return these attributes, you must implement a SCOPE_BASE search. Static internal nodes do not need SCOPE_ONELEVEL searches. The superagent performs these searches.
- Dynamic internal nodes must support SCOPE_BASE and SCOPE_ONELEVEL searches.
- Leaf nodes must support SCOPE_BASE and SCOPE_ONELEVEL searches. The DEsearch() function for a leaf node must check the node's RDN instead of the search scope to determine the type of search to perform.
If the RDN is not null, then perform a SCOPE_BASE for the object that is represented by the RDN; otherwise, enumerate all objects.

Example

The following examples demonstrate searches on static internal nodes, dynamic internal nodes, and leaf nodes.

Static Internal Nodes

The following example demonstrates a search on a static internal node:

```
int CSampleStaticInternalNode::DEsearch(DMOSearchOp *pSearchOp)
{
    LogFA(getLogID(), LOG_INFO, "CSampleStaticInternalNode::DEsearch()");

    // get the node's relative distinguished name
    DistinguishedName* pSearchDN = pSearchOp->GetDN();
    UTF8* pszuRdnValue = pSearchDN->getRDNvalue();

    int rc = LDAP_SUCCESS;
    if (pSearchOp->GetScope() <= 0) {
        // Base Level Search
        DMO_LDAP_Entry node_entry(this);

        // Retrieve the data for the node, pszuRdnValue, from the target
        // system and return its attributes to the caller.
        if <error> {
            rc = <convert the error from your target system to LDAP error
code>

            // Error message must be in UTF-8
            pszuError = "My Base Search error message";
            pSearchOp->SetStatus(pszuError, rc);

            return rc;
        }

        for <each attribute> {
            node_entry.addAttribute(<attr name>, <attr value>);
        }
        // Send the result
        rc = pSearchOp->SendEntry(&node_entry);
    }
    else {
        // Static internal node doesn't need to deal with One-Level search
        rc = LDAP_UNWILLING_TO_PERFORM;
        return rc;
    }
}
```


Dynamic Internal Nodes

The following example demonstrates a search on a dynamic internal node:

```
int CSampleInternalNode::DEsearch(DMOSearchOp *pSearchOp)
{
    DMO_LDAP_Entry *pEntry      = NULL;
    DMO_LDAP_Request *pRequest   = NULL;
    UTF8             *pszError    = NULL;
    UTF8             *pszRdnValue = NULL;
    int               rc          = LDAP_SUCCESS;

    LogFA(getLogID(), LOG_INFO, "CSampleInternalNode::DEsearch()");

    pRequest = pSearchOp->GetRequest();

    // get the node's relative distinguished name
    pszRdnValue = pDN->getRDNvalue();

    if (pRequest->getSearchScope() <= 0) {
        // Base Level Search
        pEntry = new DMO_LDAP_Entry(this);

        // Retrieve the data for the node, rdn, from the target
        // system and return its attributes to the caller.
        if <error> {
            rc = <convert the error from your target system to LDAP error
code>

            // Error message must be in UTF-8
            pszError = "My Base Search error message";
            pSearchOp->SetStatus(pszError, rc);
        }
    }
}
```

```
// Free the memory allocated for the entry.
    delete pEntry;
    pEntry = NULL;

    return rc;
}
for <each attribute> {
    pEntry->addAttribute(<attr name>, <attr value>);
}
}
else {
    // One-Level Search
    if <error> {
        rc = <convert the error from your target system to LDAP error
code>

        // Error message must be in UTF-8
        pszuError = "My One-Level Search error message";
        pSearchOp->SetStatus(pszuError, rc);

        return rc;
    }

    // Retrieve data from the target system and return the
    // data by creating an DMO_LDAP_Entry for each piece of
    // data and linking the DMO_LDAP_Entry together.
    DMO_LDAP_Entry *pCurE = NULL;
    DMO_LDAP_Entry *pPrevE = NULL;
    for <each piece of data, curData> {
        pCurE = new DMO_LDAP_Entry(this, <curData's DN>);
        if (pPrevE != NULL)
            pPrevE->setNext(pCurE);
        else
            pEntry = pCurE;
        pPrevE = pCurE;
    }
}

if (pEntry != NULL) {
    pSearchOp->SetEntry(pEntry);
}

return LDAP_SUCCESS;    // Success
}
```

Leaf Nodes

The following example demonstrates a search on a leaf node:

```
int CSampleLeafNode::DEsearch(DMOSearchOp *pSearchOp)
{
    DMO_LDAP_Entry *pEntry          = NULL;
    DMO_LDAP_Request *pRequest       = NULL;
    UTF8 *pszError                  = NULL;
    UTF8 *pszRdnValue               = NULL;
    int rc                          = LDAP_SUCCESS;

    LogFA(getLogID(), LOG_INFO, "CSampleLeafNode::DEsearch()");

    // get the node's relative distinguished name
    pszRdnValue = pDN->getRDNvalue();

    if (pszRdnValue != NULL) {
        // Base Level Search
        <do your base search here and put the data into pEntry>
        if <error> {
            rc = <convert the error from your target system to LDAP error code>
            // Error message must be in UTF-8
            pszError = "My Base Search error message";
            pSearchOp->SetStatus(pszError, rc);
            return rc;
        }
    }
    else {
        // One-Level Search
        <do your one-level search here and put the data into pEntry>

        if <error> {
            rc = <convert the error from your target system to LDAP error code>
            // Error message must be in UTF-8
            pszError = "My One-Level Search error message";
            pSearchOp->SetStatus(pszError, rc);

            return rc;
        }
    }

    if (pEntry != NULL) {
        pSearchOp->SetEntry(pEntry);
    }

    return LDAP_SUCCESS;    // Success
}
```


Appendix E: ODBC Wrapper Classes and Methods

The ODBC Wrapper class depends on the following classes:

- **AttributeMap**-Used for the C++ array representing the data map of the target SQL table.
- **AttributeMapIF**-Manipulates the C++ array of AttributeMap records.
- **CTokenEx**-Provides split/join methods to build SQL statements.
- **CTableObject**-Provides Pack/Unpack methods to map large results set from secondary tables into a multi-valued attribute.

The methods in the ODBC Wrapper class are divided into the following groups:

- Constructor
- SQL Operations
- Utility

This section contains the following topics:

[AttributeMap Class](#) (see page 302)

[AttributeMapIF Class](#) (see page 304)

[CTokenEx Class](#) (see page 314)

[CTableObject Class](#) (see page 316)

[Constructor Methods](#) (see page 318)

[SQL Operations Methods](#) (see page 318)

[Utility Methods](#) (see page 322)

AttributeMap Class

The AttributeMap class is used as a record structure in a C array. The records must be in the SQL table column order. Virtual attribute, if present, must follow all column records. The AttributeMap class is defined as follows:

```
class AttributeMap
{
public:
    UTF8* pszuLdapAttrName; // ETA LDAP attribute name
    UTF8* pszuSqlColumnName; // ODBC SQL column name
    int   iSqlType;         // column SQL data type
    bool  fMapToSql;        // Map from LDAP to SQL?
    bool  fMapToLdap;       // Map from SQL to LDAP?
    bool  fPrimaryKey;      // Primary key for the table
};
```

pszuLdapAttrName

eTrust Admin 2.0 LDAP name of the attribute.

pszuSqlColumnName

ODBC column name.

iSqlType

SQL data type of the column, as defined in the SQL data type codes section of Microsoft's SQL.H header file. The currently supported types are:

- SQL_CHAR
- SQL_DECIMAL
- SQL_INTEGER

Note: Supporting additional types requires extending the ODBC Wrapper::getLdapValueFromSqlColumn() and ODBC Wrapper::getSqlValueForSqlColumn() methods.

fMapToSq

Boolean value indicating whether this eTrust Admin 2.0 attribute is mapped to the corresponding ODBC column.

fMapToLdap

Boolean value indicating whether this ODBC column is mapped to the corresponding eTrust Admin 2.0 attribute.

fPrimaryKey

Boolean value indicating whether this ODBC column is the primary index key for retrieval of user account data.

Note: For each column in the SQL table (including the ones that are not used), an AttributeMap class object must be filled. The order of the columns must be the same as the column order of the SQL create table statement.

Encoding Virtual Attribute Information

When generating additional attributes from SQL results, or when the mapping from LDAP to SQL is not straightforward, virtual attributes can be used to provide additional mappings. Any virtual attribute must be added after all the column records are in the AttributeMap array and before the encoding of the table information.

Note: Additional code is required to actually perform the mapping.

pszLdapAttrName

eTrust Admin 2.0 LDAP name of the attribute.

pszSqlColumnName

Always NULL.

iSqlType

Always -1

fMapToSql

Boolean value indicating whether this eTrust Admin 2.0 attribute is mapped to SQL. If true, supporting code must be added to the object's DMODirectoryEntry to transform the content of this attribute to the appropriate SQL table/column(s).

fMapToLdap

Boolean value indicating whether this eTrust Admin 2.0 attribute is to be generated when receiving data from SQL. If true, supporting code must be added to the object's DMODirectoryEntry class to transform SQL data into the appropriate LDAP value.

fPrimaryKey

Always false.

Encoding Table Information

The lastAttributeMap object of the array encodes the SQL table information, as follows:

pszLdapAttrName

Always NULL.

pszSqlColumnName

ODBC SQL table name.

iSqlType

C index of the primary naming column in this array.

fMapToSql

Always false.

fMapToLdap

Always false.

fPrimaryKey

Always false.

AttributeMapIF Class

All the methods of the AttributeMapIF are available from ODBC Wrapper. These methods manipulate arrays of AttributeMap records.

getSqlRowColumnCount

Purpose

Implementation method. Returns the number of columns in the SQL table specified by the AttributeMap array specified in the ODBC Wrapper constructor.

Syntax

```
int getSqlRowColumnCount(void);
```

Arguments

None.

Return Value

Integer specifying the number of columns.

getSqlTableName

Purpose

Special method. Returns the name of the table.

Syntax

```
std::string getSqlTableName(void);
```

Arguments

None.

Return Value

C++ STL std::string object containing the name of the table.

getSqlKeyColumnIndex

Purpose

Special method. Return the index into the AttributeMap array of the primary key column.

Syntax

```
int getSqlKeyColumnIndex(void);
```

Arguments

None.

Return Value

Integer that points to the AttributeMap array.

getSqlKeyColumnName

Purpose

Special method. Returns the name of the primary key column.

Syntax

```
std::string getSqlKeyColumnName(void);
```

Arguments

None.

Return Value

C++ STL std::string containing the name of the primary key column.

getLdapAttrIndex

Purpose

Index method. Returns the index in the AttributeMap array for the specified ETA attribute name.

Syntax

```
int getLdapAttrIndex(  
    const std::string sLdapAttrName,  
    const bool fMapIndex = true  
);
```

Arguments

sLdapAttributeName

The name of the target LDAP attribute.

fMapIndex

If true, the flag indicates that the resulting index is to be used to point into the AttributeMap array.

If false, it indicates that the resulting index will be used in a vector/list representation of row data suitable for use by the ODBC Wrapper SQL operation methods.

Return Value

-1 when the attribute is not found.

getSqlColumnIndex

Purpose

Index method. Returns the index into the AttributeMap array for the specified SQL column name.

Syntax

```
int getSqlColumnIndex (  
    const std::string sSqlColumnName,  
    const bool fMapIndex = true  
);
```

Arguments

sSqlColumnName

Name of the target SQL column.

fMapIndex

Boolean:

If true, the flag indicates that the resulting index is to be used to point into the AttributeMap array.

If false, it indicates that the resulting index will be used in a vector/list representation of row data suitable for use by the ODBC Wrapper SQL operation methods.

Return Value

-1 when the attribute is not found.

getLdapAttrName

Purpose

Access method. Returns the ETA LDAP attribute name (pszLdapAttrName) of the AttributeMap record specified by the index.

Syntax

```
std::string getLdapAttrName(  
    int iMapIndex  
);
```

Arguments

iMapIndex

Index in the AttributeMap array.

Return Value

Returns one of the following:

- The corresponding LDAP attribute name in a C++ STL std::string
- An empty string ("") when the index is not valid

getSqlColumnName

Purpose

Access method. Returns the SQL column name (pszSqlColumnName) of the AttributeMap record specified by the index.

Syntax

```
std::string getSqlColumnName (  
    int iMapIndex  
);
```

Arguments

iMapIndex

Index in the AttributeMap array.

Return Value

Returns one of the following:

- The corresponding SQL attribute name in a C++ STL std::string
- An empty string ("") when the index is not valid

getSqlType

Purpose

Access method. Returns the SQL data type (iSqlType) of the AttributeMap record specified by the index.

Syntax

```
int getSqlType (  
    int iMapIndex  
);
```

Arguments

iMapIndex

Index in the AttributeMap array.

Return Value

-1 when the index is not valid.

isMappedToSql

Purpose

Access method. Returns the fMapToSql flag of the AttributeMap record specified by the index.

Syntax

```
bool isMappedToSql (  
    int iMapIndex  
);
```

Arguments

iMapIndex

Index in the AttributeMap array.

Return Value

False when the index is not valid.

isMappedToLdap

Purpose

Access method. Returns the fMapToLdap flag of the AttributeMap record specified by the index.

Syntax

```
bool isMappedToLdap (  
    int iMapIndex  
);
```

Arguments

iMapIndex

Index in the AttributeMap array.

Return Value

False when the index is not valid.

isPrimaryKey

Purpose

Access method. Returns the fPrimaryKey flag of the AttributeMap record specified by the index.

Syntax

```
bool isPrimaryKey (  
    int iMapIndex  
);
```

Arguments

iMapIndex

Index in the AttributeMap array.

Return Value

False when the index is not valid.

getLdapAttrFromSqlColumn

Purpose

Mapping method. Returns the ETA LDAP attribute name corresponding to the SQL column name.

Syntax

```
std::string getLdapAttrFromSqlColumn(  
    const std::string sSqlColumnName  
);
```

Arguments

sSqlColumnName

Column name to map to LDAP.

Return Value

Returns one of the following:

- The corresponding LDAP attribute name in a C++ STL std::string
- An empty string ("") when there is no mapping

getSqlTypeBySqlColumn

Purpose

Mapping method. Returns the SQL data type corresponding to the SQL column name.

Syntax

```
int getSqlTypeBySqlColumn (  
    const std::string sSqlColumnName  
);
```

Arguments

sSqlColumnName

Target column name.

Return Value

-1 when there is no mapping.

isSqlColumnMappedToLdap

Purpose

Mapping method. Returns the fMapToLdap flag corresponding to the SQL column name.

Syntax

```
bool isSqlColumnMappedToLdap (  
    const std::string sSqlColumnName  
);
```

Arguments

sSqlColumnName

Target column name.

Return Value

False when there is no mapping.

getSqlColumnFromLdapAttr

Purpose

Mapping method. Returns the SQL column name corresponding to the ETA LDAP attribute name.

Syntax

```
string getSqlColumnFromLdapAttr (  
    const std::string sLdapAttrName  
);
```

Arguments

sLdapAttrName

ETA LDAP attribute name to map to SQL.

Return Value

Returns one of the following:

- The corresponding SQL column name in a C++ STL std::string
- An empty string ("") when there is no mapping

getSqlTypeByLdapAttr

Purpose

Mapping method. Returns the SQL data type corresponding to the ETA LDAP attribute.

Syntax

```
int getSqlTypeByLdapAttr (  
    const std::string sLdapAttrName  
);
```

Arguments

sLdapAttrName

Target ETA LDAP attribute name.

Return Value

Returns one of the following:

- The corresponding SQL data type
- -1 when there is no mapping

isLdapAttrMappedFromSql

Purpose

Mapping method. Returns the fMapToLDAP flag corresponding to the ETA LDAP attribute. Indicates that the named LDAP attribute takes its value from an SQL column.

Syntax

```
bool isLdapAttrMappedFromSql (  
    const std::string sLdapAttrName  
);
```

Arguments

sLdapAttrName

ETA LDAP attribute name to map to SQL.

Return Value

Returns one of the following:

- True if the LDAP attribute should be populated from its corresponding SQL column
- False when there is no mapping or if the value should not be mapped from SQL

isLdapAttrMappedToSql

Purpose

Mapping method. Returns the fMapToSql flag corresponding to the ETA LDAP attribute. Indicates that the named LDAP attribute sets an SQL column.

Syntax

```
bool isLdapAttrMappedToSql (  
    const std::string sLdapAttrName  
);
```

Arguments

sLdapAttrName

Target ETA LDAP attribute name.

Return Value

Returns one of the following:

- True if the value of the named LDAP attribute should be assigned to the corresponding SQL column
- False when there is no mapping or if the value should not be mapped to SQL

CTokenEx Class

This class has the following implementations:

- Implementation that uses STL for the Agent code.
- Implementation that uses MFC for the GUI plug-in.

Split

Purpose

Splits a C++ STL `std::string` or MFC `CString` into an STL `std::string` vector (Agent) or MFC `CStringArray` (GUI) according to the Delimiter.

Syntax

STL Version:

```
void Split(  
    std::string sSource,  
    std::string sDelimiter,  
    std::vector<std::string>& AddIt,  
    BOOL fAddEmpty  
);
```

MFC Version:

```
void Split(  
    CString csSource,  
    CString csDelimiter,  
    CStringArray& AddIt,  
    BOOL fAddEmpty  
);
```

Arguments

sSource/csSource

Source string to be Split.

sDelimiter/csDelimiter Delimiter.

AddIt (Referenced)

String vector/array to which to add.

fAddEmpty

If true, add empty strings for the delimiters found.

Return Value

None.

Join

Purpose

Joins a c++ STL `std::string` vector/list or an MFC `CStringArray` to create an STL `std::string` or MFC `CString` according to the Deliminator.

Syntax

STL Versions:

```
std::string Join(
    std::string sDeliminator,
    std::vector<std::string>& AddIt
);
std::string Join(
    std::string sDeliminator,
    std::list<std::string>& AddIt
);
```

MFC Version:

```
CString Join(
    CString csDeliminator,
    CStringArray& AddIt
);
```

Arguments

sDeliminator/csDeliminator Deliminator.

AddIt (Referenced)

String vector/list or `CStringArray` containing the elements to join.

Return Value

The joined string.

CTableObject Class

This class has the following implementations:

- Implementation that uses C++ STL for the Agent code

```
std::vector<std::string>    m_Data;
```

- Implementation that uses MFC for the GUI plug-in. This class depends on the `CTokenEx` class. The class has one public variable, which holds the unpacked data:

```
CStringArraym_Data;
```

UTFPack/MBCSPack

Purpose

Packs the m_Data std::string vector/CString array into a single UTF8/MBCS string using the '|' delimiter.

Syntax

STL Version:

```
std::string UTFPack(void);
```

MFC Version:

```
CString MBCSPack(void)
```

Arguments

None.

Return Value

STL std::string or MFC CString containing the packed data from m_Data.

UTFUnpack/MBCSUnpack

Purpose

Unpacks the C++ STL std::string or MFC CString into the m_Data string vector/CString. The delimiter is '|'.

Syntax

STL Version:

```
void UTFUnpack(  
    std::string sData  
);
```

MFC Version:

```
void MBCSUnPack(  
    CString csData  
);
```

Arguments

sData/csData

C++ STL std::string or MFC CString containing the packed data.

Return Value

None.

The m_Data class variable contains the values packed in the string argument.

Constructor Methods

The Constructor group contains methods that are used to build and maintain ODBC Wrapper objects.

OdbcWrapper

Purpose

Creates an instance of the ODBC Wrapper class.

Syntax

```
OdbcWrapper(  
    OdbcDirectory      *pDir,  
    const std::string sName,  
    AttributeMap       pAttrMap[],  
    long int           iLogID;  
);
```

Arguments

***pDir**

Pointer to an OdbcDirectory object. The OdbcDirectory object has the ODBC access methods needed by OdbcWrapper to connect to the database.

sName

The Provisioning Server name of the current LDAP entry or empty.

pAttrMap

Pointer to the AttributeMap data-map array. This array should be the one representing the table that maps to the current LDAP object.

iLogID

logID so that OdbcWrapper can use the ETA2 logging facility. When the constructor is called from a DMODirectoryEntry derived class, use getLogID().

Return Value

None.

SQL Operations Methods

The SQL operations group contains methods that are called directly by the DMODirectoryEntry class for each node requiring ODBC access in your DIT.

SqlInsertRow

Purpose

The SqlInsertRow method appends a new row to the table specified in the constructor.

Syntax

```
int SqlInsertRow (  
    std::vector<std::string>    columnData  
);
```

Arguments

columnData

A C++ STL vector of strings containing column data to be added to the table. The vector is indexed by column according to the AttributeMap array.

Return Value

LDAP error code.

The value LDAP_SUCCESS represents success.

Any other value represent an error. The getErrorMessage() method can be used to retrieve the SQL error from the ODBC interface.

SqlDeleteRow

Purpose

The SqlDeleteRow method removes one or more rows from the table.

Syntax

```
int SqlDeleteRow(  
    std::vector<std::string>    columnData,  
    bool                        fSingleRow  
);
```

Arguments

columnData

A C++ STL vector of strings containing column data to be added to the table. The vector is indexed by column according to the AttributeMap array. Only the value associated with the primary key column is needed for the method to function.

fSingleRow

Indicates whether only one row is expected to be deleted. This is to catch a problem occurring when more than one record with a given primary key is stored in the table.

Return Value

LDAP error code.

The value LDAP_SUCCESS represents success.

If fSingleRow is true and more than one row is affected, the error code LDAP_OPERATIONS_ERROR is returned.

If no row is deleted, LDAP_NO_SUCH_OBJECT is returned.

The getErrorMessage() method can be used to retrieve the SQL error from the ODBC interface.

SqlUpdateRow

Purpose

The SqlUpdateRow method modifies a single row in a table.

Syntax

```
int SqlUpdateRow (  
    std::vector<std::string>    columnData  
);  
  
int SqlUpdateRow (  
    std::vector<rowDataEntry>    columnData  
);
```


Arguments

columnData Form 1

A C++ STL vector of strings containing column data to update the row. The vector is indexed by column according to the AttributeMap array. The value associated with the primary key column must be present. The remaining vector items will replace the original values in the row.

columnData Form 2

A C++ STL vector of rowDataEntry class objects, one object per column.

The rowDataEntry class is defined as follows:

```
class rowDataEntry (
{
public:
    std::string      sValue;
    /* iOperation values -- from ldap.h
    #define LDAP_MOD_ADD          ((ber_int_t) 0x0000)
    #define LDAP_MOD_DELETE ((ber_int_t) 0x0001)
    #define LDAP_MOD_REPLACE ((ber_int_t) 0x0002)
    */
    int              iOperation;
};
```

This form permits the pass-through of LDAP modify information on each specific column. With form 1, the value associated with the primary key column must be present.

Return Value

LDAP error code.

The value LDAP_SUCCESS represents success.

The value LDAP_NO_SUCH_OBJECT indicates that the row specified by the primary key column was not found.

The value LDAP_OPERATIONS_ERROR indicates another error.

The getErrorMessage() method can be used to retrieve the SQL error from the ODBC interface.

executeSqlUpdate

Purpose

The executeSqlUpdate method sends an SQL update query command to the SQL database, which does not return a result. This method is used by the SqlDeleteRow and SqlUpdateRow methods.

Syntax

```
int executeSqlUpdate (
    const std::string  sSqlStatement,
    int                *piUpdatedRows
);
```

Arguments

sSqlStatement

SQL update query statement.

piUpdatedRows

Pointer to an integer to store the number of rows updated.

Return Value

LDAP error code.

The value LDAP_SUCCESS represents success.

The value LDAP_CONNECTION_ERROR will be returned if the method was unable to reach the ODBC target or an error occurred on the SQL statement.

Additional information can be retrieved by calling the getErrorMessage() method.

Utility Methods

The utility group contains supporting methods used by the SQL operations group, and may be useful for customizing the behavior of the ODBC agent.

getLdapValueFromSqlColumn

Purpose

The getLdapValueFromSqlColumn method returns the LDAP value from specified column of the given result set. It performs the appropriate data type conversion from the SQL value to LDAP.

This method must be modified and enhanced if additional SQL data types need to be supported.

The SQL data types are defined in the SQL data type codes section of Microsoft's SQL.H header file. The currently supported types are:

- SQL_CHAR
- SQL_DECIMAL
- SQL_INTEGER

Syntax

```
std::string getLdapValueFromSqlColumn (  
    ResultSet          *pResultSet,  
    const std::string  sSqlColumnName  
);
```

Arguments

pResultSet

Pointer to a result set from a Odbc::Statement::executeQuery() call.

sSqlColumnName

Name of the column to retrieve from the result set.

Return Value

The LDAP value (in a C++ STL std::string) in the column identified by sColumnName in the result set, pResultSet.

getSqlValueForSqlColumn

Purpose

The getSqlValueForSqlColumn method returns a valid SQL value for the given LDAP value, as specified by the data type associated with the specified SQL column. It performs the appropriate data type conversion from an LDAP string to an SQL.

This method must be modified and enhanced if additional SQL data types need to be supported.

The SQL data types are defined in the SQL data type codes section of Microsoft's SQL.H header file. The currently supported types are:

- SQL_CHAR
- SQL_DECIMAL
- SQL_INTEGER

Syntax

```
std::string getSqlValueForSqlColumn (  
    const std::string      sLdapValue,  
    const std::string      sSqlColumnName  
);
```

Arguments

sLdapValue

The LDAP value to convert.

sSqlColumnName

The SQL column to which LDAP value will be mapped.

Return Value

A valid SQL value encoded in a C++ STL std::string.

Comments

The returned SQL value will have single quotes escaped.

escapeQuotes

Purpose

The escapeQuotes static method escapes single quotes to create a valid SQL string literal.

Syntax

```
static std::string escapeQuote (  
    const std::string  sUnescaped  
);
```

Arguments

sUnescaped

String with single quotes not yet escaped.

Return Value

A new C++ STL std::string with the single quotes, if any, escaped.

Comments

The input string is not changed.

getErrorMessage

Purpose

The getErrorMessage method returns the current error message.

Syntax

```
std::string getErrorMessage (void);
```

Arguments

None.

Return Value

The error message associated with the last call to other OdbcWrapper class methods.

getTableRowCount

Purpose

The getTableRowCount API returns the number of rows that have the primary key set to the value provided in the OdbcWrapper's constructor. A second form of the method is available to permit searches for specific column values.

Syntax

```
int getTableRowCount (
    int                                     *piNumRows
);
int getTableRowCount (
    std::vector<std::string>               columnData,
    int                                     *piNumRows
);
```

Arguments

****piNumRows***

Pointer to an integer to return the number of rows.

columnData

STL vector of strings of the same size as the table being manipulated. Use the vector positions that have a string value to test for a match against the corresponding column from the table.

Return Value

LDAP error code.

The value LDAP_SUCCESS represents success.

Additional information can be retrieved by calling the getErrorMessage() method.

getPackedRows

Purpose

The getPackedRows method provides a facility to encode data returned from queries returning multiple rows. Each row is packed into a string that can be passed by LDAP and unpacked by the receiver without losing column information. Optionally, getPackedRows can be sorted by the specified columns using the SQL ORDER BY clause.

This method is most useful when secondary account data is stored in additional tables.

The SQL query is hard-coded to return all rows for the primary key specified in the constructor.

Syntax

```
int getPackedRows (
    std::list<std::string>&    rowList
);
int getPackedRows (
    std::list<std::string>&    rowList,
    std::list<std::string>     lsOrderByColumns
);
```

Arguments

rowList

STL list of strings, each item of the list is a packed row.

lsOrderByColumns

STL list of column names to order by. The order of the values determines the ordering precedence.

Return Value

LDAP error code.

The value LDAP_SUCCESS represents success.

Additional information can be retrieved by calling the getErrorMessage() method.

Comment

The CTableObject class provides the facilities to pack and unpack the packed rows for both the Agent and GUI projects.

Appendix F: The Java IAM SDK

The Java Identity and Access Management (JIAM) SDK does the following:

- Provides a Java interface to the Provisioning Server
- Provides software development groups with an easy-to-use abstraction of the Provisioning Server functionality to rapidly develop custom client applications
- Acts as the single interface to supply multiple clients with access to Identity and Access Management functionality

This section contains the following topics:

[Installing the JIAM SDK](#) (see page 329)

[Setting Up the JIAM SDK](#) (see page 330)

[Sample Programs](#) (see page 330)

[JIAM SDK Reference](#) (see page 330)

Installing the JIAM SDK

Perform the following procedure to install the Provisioning Server JIAM SDK from your CA Identity Manager installation media.

To install the JIAM SDK

1. Locate the CA Identity Manager Provisioning Components download or other media.
2. Start the Product Explorer.
3. Select Install Products, Developer Resources.
4. Select JIAM SDK.
5. Follow the onscreen instructions to complete the installation.

Note: The default installation path for the JIAM SDK is C:/Program Files/CA/eTrust JIAM SDK.

Setting Up the JIAM SDK

To develop applications using JIAM SDK, do one of the following:

- Include all the jar files in the /lib directory in the classpath.
- Include /lib/jiam.jar in the jar manifest classpath of your application.

For more information, see:

<http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html>.

Sample Programs

JIAM SDK installs the SDKSample.java file into the /examples folder. This file contains a sample application that uses the JIAM SDK.

By default, the sample application attempts to connect to the Provisioning Server on the local machine. To run the sample application, edit the SDKSample.properties file if needed. If your Provisioning Server is on a different machine, change the "com.ca.iam.server.name" property appropriately.

The "username," "password," and "domainName" properties should also be changed to match a user on your Provisioning Server. This user should have sufficient rights to search for users and create new users.

To launch the sample application, use run.bat for Windows and run.sh for UNIX.

JIAM SDK Reference

For information on the JIAM SDK, see the */doc/api/index.html* file in the installation folder.

Appendix G: Custom Connectors

Existing Provisioning Server implementations can manage custom connectors, or endpoint types. To enable Identity Manager to read custom endpoint policies and accounts that are associated with provisioning roles, there are additional procedures required. This section describes the concepts and the required procedures for custom endpoint types.

This section contains the following topics:

[The JIAM Custom Connector](#) (see page 331)

[How to Define JIAM Custom Connector Extensions](#) (see page 332)

[How to Deploy JIAM Custom Connector Jar Files](#) (see page 333)

[How to Validate Custom Connectors](#) (see page 334)

The JIAM Custom Connector

Identity Manager uses the Provisioning Server JIAM API to manage provisioning roles, to synchronize Identity Manager users with accounts, and to view user accounts. The JIAM API can also enable Identity Manager to manage custom objects.

To support custom endpoint objects, the JIAM API must have the following:

- A JIAM property that describes where the JIAM API can load object metadata for the custom connector.
- An application server-defined location that includes the option-specific jar that contains all the required java objects for the custom connector.

You develop custom connectors using the Provisioning SDK. In the Provisioning SDK, there is a build step that creates an connector-specific jar, which contains the necessary object implementations for managing that custom connector through the JIAM API. For each connector, there is a separate jar file.

Note: The connector jars are not required when deploying the connector to Provisioning Server or Provisioning Manager hosts. Therefore, custom connectors may not have existing jar files. Each Identity Manager instance requires these connector-specific jars to read the connector's custom objects.

How to Define JIAM Custom Connector Extensions

Each custom connector contains an `ExtensionDescriptor` object that defines the metadata necessary for use with the JIAM API. This object is packaged in the connector's jar file. To configure JIAM to support a custom object, you specify the `ExtensionDescriptor` object's class name in a `JIAMExtensions` property when JIAM is initialized by Identity Manager.

In the Management Console, define the `JIAMExtensions` property in the Miscellaneous Properties (in Advanced Settings) for an Identity Manager environment. Specify the property as follows:

Property Name: `JIAMExtensions`
Description: Defines available custom connector extensions
Value: Fully-qualified `ExtensionDescriptor` class name.

Note: To add support for multiple custom connectors, specify the `ExtensionDescriptor` class name for each connector in a space-delimited list.

For example, the Provisioning SDK includes a sample connector called "SDK." When you build this sample, the optional build step creates a jar file named `jamExtSDK.jar`. The fully qualified name of the `ExtensionDescriptor` object within that jar is:

```
com.ca.iam.model.options.sdk.impl.SDKOptionDescriptor
```

To use this connector in an Identity Manager environment, create the `JIAMExtensions` property as follows:

Property `JIAMExtensions`
:
Value: `com.ca.iam.model.options.sdk.impl.SDKOptionDescriptor`

Note: Do not add the `.class` suffix to the value.

Identity Manager deployments that require custom connector objects must define this property for each of the custom connectors they want to manage in the Identity Manager environment. This applies to Identity Manager environments that directly manage a Provisioning Server and a Provisioning Directory, and to environments that include a CA Identity Manager user store and a provisioning directory.

How to Deploy JIAM Custom Connector Jar Files

For Identity Manager to manage custom connectors, each `ExtensionDescriptor` object defined in the `JIAMExtensions` list, and the associated custom connector jar must be defined in the classpath. Since Identity Manager runs in an application server, each JIAM extension jar must be present on an application server-specific classpath.

Deployment on WebLogic or WebSphere

For WebLogic and WebSphere application servers, you have to extract the connector jar file before the application server can use it.

To deploy a custom connector jar file on WebLogic or WebSphere

1. Extract the contents of each connector jar file to the following Identity Manager EAR folder:
 - For WebLogic:
`weblogic_home/weblogic_domain/Identity Minder.ear/custom`
 - For WebSphere:
`websphere_home/installedApps/server_Instance/Identity Minder.ear/custom`
2. Restart the application server.

Deployment on JBoss

JBoss application servers automatically load any jar files that are placed in the `jboss_home\server\default\lib` directory. You can use this feature to load a custom connector jar file.

To deploy custom connector jars on Jboss

1. Copy each connector jar file to the `jboss_home\server\default\lib` directory.
2. Restart the application server.

How to Validate Custom Connectors

To validate that each custom connector is initialized properly, define an additional miscellaneous property in the Management Console to perform a validation step. This property and its values are shown below:

Property Name:	JIAMExtensionsValidate
Description:	Validates that Identity Manager can load each of the JIAM Extension Descriptor objects defined in the JIAMExtensions property
Values:	■ True: Attempts to instantiate each of the defined descriptor objects. If the object can not be loaded, it is not added to the list of JIAM Extensions. The custom connector objects can not be used. ■ False: No validation occurs. Instantiation of the descriptor object is delayed until a reference to any custom connection object is made. Failures at run-time may prevent use of certain tasks until this condition is resolved.

Note: For performance reasons, you should set the value of this property to 'False' after validating the JIAM extensions.

To troubleshoot any custom connector load failures, set the following Debug category in the IdentityMinder.ear\config\com\netegrity\config\log4j _<app server>.properties file:

```
log4j.category.ims.llsdk.etrustadmindirectory=DEBUG
```

```
log4j.category.ims.llsdk.etrustadmindirectory=true
```

Appendix H: Universal Provisioning Connector Reference

This section contains the following topics:

[UPC Account Objects](#) (see page 335)

[Program Exit Types](#) (see page 340)

[Non-Managed Mode Operations](#) (see page 357)

UPC Account Objects

The properties of the attributes of the Universal Provisioning Account class include the following:

- Account Properties
- Operational Properties
- Virtual Properties
- Deprecated Properties

Account Properties

eTUPOAccountName

The name of the account.

eTUPOUserData

The user data associated with the account. The program exits, both common and UPC, are expected to understand the structure of this data if they need access to specific elements. The format of the data in this property is free-form. The maximum length for the user data is 2048 characters.

Operational Properties

The following properties are stored in the repository when UPC operates in the non-managed mode.

eTUPOAccountStatus

Indicates the current status of the account. The value is a single letter:

A—active account

D—adeleted account

P—an account with an operation pending

R—an account marked for physical removal from the repository upon the next receipt of a DELETE request.

eTUPORootOperationID

The Provisioning Server ID of the top-level operation which called upon UPC for a user-provisioning operation. The format is as generated by Microsoft's Platform SDK: Remote Procedure Call (RPC) UuidToString() function.

eTUPOOperationID

A string generated by the UPC SASP Agent plug-in to uniquely identify the current operation. The format is as generated by Microsoft's Platform SDK: Remote Procedure Call (RPC) UuidToString() function.

eTUPOOperationType

Indicates the operation type. The value is a single integer:

0—Add operation

1—Delete operation

2—Modify operation (includes password, enable/disable operations)

3—Rename operation

eTUPOOperationTimeStamp

Date and time of the operation using the time_t format. The value is an integer. Used by the SLA Monitor Service.

eTUPOOperationDate

Date of the operation, in Unicenter Date format. Unused directly.

eTUPOOperationTime

Time of the operation, in Unicenter Time format. Unused directly.

eTUPOOperationHistory

Log of all the user-provisioning operations requested on this account. Each value is a pipe-delimited (|) list of operational data, structured as follows:

eTUPOOperationDate

From the corresponding attribute when the account was marked as "completed."

eTUPOOperationTime

From the corresponding attribute when the account was marked as "completed."

eTUPOOperationType

From the corresponding attribute when the account was marked as "completed."

eTUPOOperationID

From the corresponding attribute when the account was marked as "completed."

Operation Status

Always 'C' for completed.

Operation Completed Date

Date on which the mark "completed" was received, in Unicenter Date format

Operation Completed Time

The time at which the mark "completed" was received, in Unicenter Date format

Operation Completed Note

An optional user-defined string that was provided when the mark "completed" operation was sent.

The UPC SASP agent plug-in takes the existing operation properties from the entry when it is marked as "completed", creates the structured records, and appends it to the history log. The following is an example:

```
0000104335|0002851800|0000000000|37f894fb-5683-441a-82e5-  
d49382918f10|C|104335|3083000|  
0000104335|0003086300|0000000002|f93cab2b-13a8-45aa-b9ec-  
fae3bfe84596|C|104335|3092200|  
0000104335|0003099900|0000000002|36952a5f-346f-493a-9a6c-  
f50834d2c9e1|C|104335|3111100|  
0000104335|0003138700|0000000002|211c1b0b-649e-4607-9fbe-  
4df6a1fa1209|C|104335|3141900|
```

Virtual Properties

The following properties are used by the UPC SASP agent plug-in.

eTUPOOperationStatus

Used to update the status of the account.

Mark the account's current pending operation as completed.

Mark the account for physical removal from the repository the next time it receives a DELETE operation request.

eTUPOOperationStatusNote

Used in conjunction with the eTUPOOperationStatus when marking an account as "completed". This property contains a user-specified string which will be added to the eTUPOOperationHistory log. The text has a maximum length of 512 characters.

eTUPOSource

Used internally by the UPC SASP agent plug-in.

Deprecated Properties

The schema definition of an eTUPOAccount object has been simplified by deprecating properties unused by the Provisioning Server agent. If your policies use these attributes, their values must be moved into the user data field (eTUPOUserData attribute). This may require corresponding changes in any UPC program exit that was developed to manipulate the corresponding elements in the Input XML buffer.

The following UPC account properties have been deprecated:

- eTUPOGlobalUserName
- eTUPOEmailAddress
- eTUPOManagerGlobalUserName
- eTUPOManagerEmailAddress

For example the r8.0 default UPC policy used to be the following:

```
objectClass          eTUPOPolicy
eTUPOPolicyName      UP0DefaultPolicy
eTDescription        UP0Default Policy
eTUPOAccountName     %AC%
eTUPOGlobalUserName  %U%
eTUPOEmailAddress    %UE%
eTUPOManagerGlobalUserName %UWFM%
eTUPOUserData        Fullname=%UN%
```

The r8.1 or r8.2 default UPC policy would be replaced by the following:

```
objectClass          eTUPOPolicy
eTUPOPolicyName      UP0DefaultPolicy
eTDescription        UP0Default Policy
eTUPOAccountName     %AC%
eTUPOUserData        Fullname=%UN%
                     GlobalUserName=%U%
                     EmailAddress=%UE%
                     ManagerGlobalUserName=%UWFM%
```

The r8.1 or r8.2 default UPC policy could also be replaced by the following:

```
objectClass                eTUPOPolicy
eTUPOPolicyName            UP0DefaultPolicy
eTDescription              UP0Default Policy
eTUPOAccountName          %AC%
eTUPOUserData              <myData>
                           <Fullname>%UN%</FullName>
                           <EmailAddress>%UE%</EmailAddress>
                           <GlobalUserName>%U%</GlobalUserName>
                           <ManagerGlobalUserName>%UWFM%</ManagerGlobalUserName>
</myData>
```

In either case, the UPC Program Exits that are called using this policy should be updated if they retrieve specific information from the user data attribute rather than from the original corresponding attributes.

Program Exit Types

Exit types determine the circumstances under which an exit is called. One of the available types of exits is entered for eTExitType (in the input XML buffer passed to the program exit).

Note: In all cases, the name of the object being passed is sent. This is formatted in both DN and Common Name format.

Common Exit Types for Managed and Non-Managed Modes

INVOCATION_ERROR

INVOCATION_ERROR is called by the Exit Interface library when it encounters a failure invoking an UPC program exit. This is for the actual invocation of the program exit, not based on the return status of the exit. This exit ignores the setting of the IgnoreFailure tag in the eTExitCustomData tag.

Input XML Buffer

The Input XML buffer will be the same as what was sent to the failed exit.

ADD_ACCOUNT

ADD_ACCOUNT is called by the Universal Provisioning Option SASP agent plug-in when it receives a request to create a new account.

Input XML Buffer

XML Operation Block

The XML Operation block is present only if invoked from a non-managed mode directory.

XML Account Block

The password information (eTPassword) is included in the XML Account block.

Return XML Buffer

eTExitReturnCategory

Set to one of SUCCESS, WARNING or FAILURE.

eTExitReturnNative

For non-managed mode, specifies the return value from the native program exit call. For managed mode, this should be a string representation of the corresponding integer LDAP error code (see ldap.h).

eTExitLogMsg

If the eTExitReturnCategory is not SUCCESS, then some sort of message should be included in this block. The message will be logged to the Provisioning Server.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

Not used.

DELETE_ACCOUNT

Called by the DELETE request.

Input XML Buffer

XML Operation Block

The XML Operation block is only present if invoked from a non-managed mode directory.

XML Account Block

The distinguished name (eTDN) and account name (eTName) are the only attributes available to this exit.

Return XML Buffer

eTExitReturnCategory

Set to one of SUCCESS, WARNING or FAILURE.

eTExitReturnNative

For non-managed mode, this specifies the return value from the native program exit call. For managed mode, this should be a string representation of the corresponding integer LDAP error code (see ldap.h).

eTExitLogMsg

If the eTExitReturnCategory is not SUCCESS, then some sort of message should be included in this block. The message will be logged to the Provisioning Server.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

Not used.

MODIFY_ACCOUNT

Called by UPC when a modify account is invoked. For non-managed mode, changes to the password or changes to the suspension status of an account are handled by the CHANGE_ACCOUNT_PASSWORD and ENABLE_ACCOUNT or DISABLE_ACCOUNT exits respectively. If either or both changes are in the modify request, the UPC will return an operation error to the requestor.

In managed mode, all changes are handled through this exit type.

Input XML Buffer

XML Operation Block

The XML Operation block is only present if invoked from a non-managed mode directory.

XML Account Block

For non-managed mode, the password (eTPassword) is excluded from the block. Managed mode exits need to parse the Account block to retrieve the elements it is interested in. Of particular note are the account status attribute eTSuspended, and the password attribute eTPassword. All attributes changes have the following format:

```
<{attributeName} modify-mode={mode}> {value} </{attributeName}>
```

{mode}

"add," "delete," or "replace," indicating whether the {value} is to be added or deleted, or should replace the existing value.

Return XML Buffer

eTExitReturnCategory

Set to one of SUCCESS, WARNING or FAILURE.

eTExitReturnNative

For non-managed mode, specifies the return value from the native program exit call. For managed mode, this should be a string representation of the corresponding integer LDAP error code (see ldap.h).

eTExitLogMsg

If the eTExitReturnCategory is not SUCCESS, then some sort of message should be included in this block. The message will be logged in the Provisioning Server.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

Not used.

RENAME_ACCOUNT

Called by the MODIFYRDN request.

Input XML Buffer

XML Operation Block

The XML Operation block is only present if invoked from a non-managed mode directory.

XML Account Block

The new account name (eTNewName) and the new distinguished name (eTNewDN) are added to the current account name (eTName) and distinguished name (eTDN) in the XML input buffer.

Return XML Buffer

eTExitReturnCategory

Set to one of SUCCESS, WARNING or FAILURE.

eTExitReturnNative

For non-managed mode, specifies the return value from the native program exit call. For managed mode, this should be a string representation of the corresponding integer LDAP error code (see ldap.h).

eTExitLogMsg

If the eTExitReturnCategory is not SUCCESS, some message should be included in this block. The message will be logged in the Provisioning Server.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

Not used.

Non-Managed Mode Exit Types

CHANGE_ACCOUNT_PASSWORD

This is a special case of a MODIFY request, which is called when the request only contains a password change.

Input XML Buffer

XML Operation Block

Present.

XML Account Block

The distinguished name (eTDN), account name (eTName) and the password (eTPassword) are the only attributes available to this exit.

The password change will look like the following:

```
<eTPassword modify-mode=replace> new password </eTPassword>
```

Return XML Buffer

eTExitReturnCategory

SUCCESS, WARNING or FAILURE.

eTExitReturnNative

Specifies the return value from the native program exit call.

eTExitLogMsg

If the value of eTExitReturnCategory is not SUCCESS, some message should be included in this block. The message will be logged to the Provisioning Server.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

Not used.

ENABLE_ACCOUNT

A special case of a MODIFY request. Called when the enable attribute of the account is to be changed. The enable attribute is when eTSuspended is set to 0.

Input XML Buffer

XML Operation Block

Present.

XML Account Block

The distinguished name (eTDN) and the account name (eTName) are the only account attributes available to this exit.

Return XML Buffer

eTExitReturnCategory

SUCCESS, WARNING or FAILURE.

eTExitReturnNative

Specifies the return value from the native program exit call.

eTExitLogMsg

If the eTExitReturnCategory is not SUCCESS, a message should be included in this block. The message is logged in the Provisioning Server.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

Not used.

DISABLE_ACCOUNT

A special case of a MODIFY request, it is called when the disable attribute of the account is to be changed.

Note: The account is disabled when eTSuspended is set to 1.

Input XML Buffer

XML Operation Block

Present.

XML Account Block

The distinguished name (eTDN) and the account name (eTName) are the only account attributes available to this exit.

Return XML Buffer

eTExitReturnCategory

SUCCESS, WARNING or FAILURE.

eTExitReturnNative

Specifies the return value from the native program exit call.

eTExitLogMsg

If the eTExitReturnCategory is not SUCCESS, a message should be included in this block. The message will be logged in the Provisioning Server.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

Not used.

REQUEST_PENDING

Called by the UPC Agent plug-in when a modify, add, or delete request successfully was sent out to a non-managed system administrator, and after UPC marked the repository's account status attribute (eTUPOAccountStatus) to pending (P).

Input XML Buffer

XML Operation Block

Present.

XML Account Block

The distinguished name (eTDN) and the account name (eTName) are the only account attributes available to this exit.

Return XML Buffer

eTExitReturnCategory

SUCCESS, WARNING or FAILURE.

eTExitReturnNative

Specifies the return value from the native program exit call.

eTExitLogMsg

If the eTExitReturnCategory is not SUCCESS, a message should be included in this block. The message will be logged in the Provisioning Server.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

Not used.

REQUEST_COMPLETED

Called by the UPC SASP agent plug-in when an external modify request successfully marked the repository's account status attribute (eTUPOAccountStatus) from pending (P) to completed (C).

Input XML Buffer**XML Operation Block**

Present.

XML Account Block

The distinguished name (eTDN) and the account name (eTName) are the only account attributes available to this exit.

Return XML Buffer**eTExitReturnCategory**

SUCCESS, WARNING or FAILURE.

eTExitReturnNative

Specifies the return value from the native program exit call.

eTExitLogMsg

If the eTExitReturnCategory is not SUCCESS, a message should be included in this block. The message will be logged in the Provisioning Server.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

Not used.

Service Level Agreement Monitor Exits (Non-Managed Mode)

The following exit types are invoked only by the SLA Monitor service. For these exits, the XML Account block of the Input XML buffer is replaced with an XML Directory block:

```
<eTUPODirectory>
  <eTDN> </eTDN>
  <eTName> </eTName>
</eTUPODirectory>
```

DN

The distinguished name of the Directory.

Name

The name of the Directory.

SLA_EXCEEDED

Called by the UPC Monitoring service when pending requests are found that exceed the SLA set for the corresponding directory.

Input XML Buffer

XML Operation Block

Present.

XML Directory Block

The distinguished name (eTDN) and the directory name (eTName) are included. The list of accounts currently exceeding the SLA limit is encoded in an Account block in the XML Directory Block:

```
<eTUPODirectory>
  <eTDN> </eTDN>
  <eTName> </eTName>
  <eTUPOAccount>
    <eTUPOAccountName> {name 1}</eTUPOAccountName>
    <eTUPOAccountName> {name 1}</eTUPOAccountName>
    <eTUPOAccountName> ...?</eTUPOAccountName>
    <eTUPOAccountName> {name n}</eTUPOAccountName>
  </eTUPOAccount>
</eTUPODirectory>
```

Return XML Buffer**eTExitReturnCategory**

SUCCESS, WARNING or FAILURE.

eTExitReturnNative

Specifies the return value from the native program exit call.

eTExitLogMsg

If the eTExitReturnCategory is not SUCCESS, a message should be included in this block. The message will be logged by the UPC Monitoring service.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

Not used.

SLA_WARNING

Called by the UPC Monitoring service when pending requests are found that exceed the SLA Warning level set for the corresponding directory. The list of accounts is encoded in the same way as the SLA_EXCEEDED exit type.

Input XML Buffer

XML Operation Block

Present.

XML Directory Block

The distinguished name (eTDN) and the directory name (eTName) are included. The list of accounts currently exceeding the warning limit is encoded in an Account block in the XML Directory Block:

```
<eTUPODirectory>
  <eTDN> </eTDN>
  <eTName> </eTName>
  <eTUPOAccount>
    <eTUPOAccountName> {name 1}</eTUPOAccountName>
    <eTUPOAccountName> {name 1}</eTUPOAccountName>
    <eTUPOAccountName> ...?</eTUPOAccountName>
    <eTUPOAccountName> {name n}</eTUPOAccountName>
  </eTUPOAccount>
</eTUPODirectory>
```

Return XML Buffer

eTExitReturnCategory

SUCCESS, WARNING or FAILURE.

eTExitReturnNative

Specifies the return value from the native program exit call.

eTExitLogMsg

If the eTExitReturnCategory is not SUCCESS, a message should be included in this block. The message is logged by the UPC Monitoring service.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

Not used.

SLA_TIMEOFFSET

Called by the UPC Monitoring service before testing for accounts exceeding the SLA or warning level. Used to skip non-business periods where the SLA may not apply. This exit must return an integer representing the number of seconds to subtract from the current time before determining if a pending operation exceeds any of the two time limits. The data is returned in the eTEExitCustom block, in the following manner:

```
<eTEExitCustom>
  <eTFuncReturn> {integer} </eTFuncReturn>
</eTEExitCustom>
```

Note: If multiple exits of this type are invoked, only the return of the last exit executed is used as the offset.

Input XML Buffer

XML Operation Block

Present.

XML Directory Block

The distinguished name (eTDN) and the directory name (eTName) are included. The list of accounts currently exceeding the SLA limit is encoded in an Account block in the XML Directory Block:

```
<eTUPODirectory>
  <eTDN> </eTDN>
  <eTName> </eTName>
  <eTUPOAccount>
    <eTUPOAccountName> {name 1}</eTUPOAccountName>
    <eTUPOAccountName> {name 1}</eTUPOAccountName>
    <eTUPOAccountName> ...?</eTUPOAccountName>
    <eTUPOAccountName> {name n}</eTUPOAccountName>
  </eTUPOAccount>
</eTUPODirectory>
```

Return XML Buffer

eTEExitReturnCategory

SUCCESS, WARNING or FAILURE.

eTEExitReturnNative

Specifies the return value from the native program exit call.

eTEExitLogMsg

If the eTEExitReturnCategory is not SUCCESS, a message should be included in this block. The message will be logged by the UPC Monitoring service.

eTEExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

Not used.

Managed-Mode Exits

READ_ACCOUNT

Called by the UPC Agent plug-in when a request to read an account is received. This exit must return the user data of the account in the eTExitCustom block of the XML return buffer

Input XML Buffer

XML Operation Block

Not present.

XML Account Block

The distinguished name (eTDN) and the account name (eTName) are the only account attributes available to this exit.

Return XML Buffer**eTExitReturnCategory**

SUCCESS, WARNING or FAILURE.

eTExitReturnNative

This should be a string representation of the corresponding integer LDAP error code (see ldap.h).

eTExitLogMsg

If the eTExitReturnCategory is not SUCCESS, a message should be included in this block. The message will be logged in the Provisioning Server.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

This block needs to include an eTFuncReturn block which contains the user data read by the exit. This information is copied word for word in the eTUPOUserData attribute.

The following is an example from the UPO-SDK:

```
<eTExitCustom>
  <eTFuncReturn>
    <eTSDKAccount>
      <eTSDKCity>Paris</eTSDKCity>
      <eTSDKEmployeeId>5</eTSDKEmployeeId>
      <eTSDKGroupMembers>staff</eTSDKGroupMembers>
      <eTSDKGroupMembers>dev</eTSDKGroupMembers>
    </eTSDKAccount>
  </eTFuncReturn>
</eTExitCustom>
```

The format used here must be followed exactly in the UPC policy which refers to these entries.

LIST_ACCOUNTS

Called by the UPC Agent plug-in when a request to enumerate accounts is received. This exit must return a list of account names.

Input XML Buffer

XML Operation Block

Not present.

XML Account Block

This block contains the distinguished name and object name values of the UPC account container rather than of an UPC account:

```
<eTUPOAccount>
  <eTDN>eTUPOAccountContainerName=Accounts,
    eTUPODirectoryName={DIRECTORY_NAME},
    eTConnectorName=Universal Provisioning,
    dc={DOMAIN_NAME},
    dc=etasa
  </eTDN>
  <eTName>Accounts</eTName>
</eTUPOAccount>
```

Return XML Buffer

eTExitReturnCategory

SUCCESS, WARNING or FAILURE.

eTExitReturnNative

This should be a string representation of the corresponding integer LDAP error code (see ldap.h).

eTExitLogMsg

If the eTExitReturnCategory is not SUCCESS, then some sort of message should be included in this block. The message will be logged in the Provisioning Server.

eTExitContinue

Specifies whether to continue the process flow after the return from the program exit.

eTExitCustom

The data is returned in this block, using the following structure:

```
<eTExitCustom>
  <eTFuncReturn> {account name 1} </eTFuncReturn>
  <eTFuncReturn> {account name 1} </eTFuncReturn>
  <eTFuncReturn> ...???</eTFuncReturn>
  <eTFuncReturn> {account name n} </eTFuncReturn>
</eTExitCustom>
```

Non-Managed Mode Operations

In non-managed mode, UPC marks an account as having a pending operation after it has successfully invoked the operation exit (if any). While an account is in the pending state, any new user-provisioning operation will immediately fail, ensuring that only one outstanding operation is permitted against an account. To clear this state, the outstanding operation must be marked as completed.

The account repository's eTUPOAccountStatus attribute is used to store this status, which is read-only.

Marking a Pending Operation As Completed

A Pending operation can be marked as completed by using the Provisioning Manager, etautil, or the LDAP protocol directly. A short note can be added when completing the operation.

To mark an operation as completed, you need to modify the eTUPOOperationStatus attribute and replace its value with the character C. Optionally, the eTUPOOperationStatusNote can include a short string, which will be added to the account's operation history log.

Using ETAUTIL

The following is an example of how to use the etautil utility to mark a pending operation as completed:

```
etautil [-d domain] [-u user [-p password]] update  
'eTUPOAccountContainerName=Accounts, eTUPODirectoryName=<DIRECTORY_NAME>,  
eTNamespaceName=Universal Provisioning ' eTUPOAccount  
eTUPOAccountName=<ACCOUNT_NAME> to eTUPOOperationStatus='C',  
eTUPOOperationStatusNote:'Completed the request. Assigned to default group  
as requested.'
```

Using an LDAP client

You can use an LDAP client to send a modify operation against the UPC account entry in the repository. This operation sets the eTUPOOperationStatus attribute to C. To have an LDAP client send a modify operation, use the global user configured in the <DIRECTORY_NAME> property sheet as the user for the Bind request.

The bind parameters are:

```
DN: eTGlobalUserName=etaupoad, eTGlobalUserContainerName=Global Users,  
eTNamespaceName=CommonObjects,  
dc=<DOMAIN_NAME>, dc=eta
```

where password is the etaupoad's password.

The modify operation detail using the LDIF format is the following:

```
dn: eTUPOAccountName=<ACCOUNT_NAME>, eTUPOAccountContainerName=Accounts,  
eTUPODirectoryName=<DIRECTORY_NAME>, eTNamespaceName=Universal Provisioning,  
dc=<DOMAIN_NAME>, dc=eta  
  
changetype: modify  
replace: eTUPOOperationStatus  
eTUPOOperationStatus: C  
-  
replace: eTUPOOperationStatusNote  
eTUPOOperationStatusNote: Completed the request. Assigned to default group as  
requested.
```

Remove an Account from the Directory

When deleting an account, the LDAP delete operation is passed on to UPC as a Delete operation request, not as an actual request to remove the account from the Directory. Removing an account from the Directory, therefore, is a two-step process.

To remove an account from the directory

1. Mark the account for deletion by modifying the account's eTUPOOperationStatus attribute by replacing its value with a single character 'R'. Using ETAUTIL, the command is as follows:

```
etautil [-d domain] [-u user [-p password]] update  
'eTUPOAccountContainerName=Accounts, eTUPODirectoryName=<DIRECTORY_NAME>,  
eTNamespaceName=Universal Provisioning ' eTUPOAccount  
eTUPOAccountName=<ACCOUNT_NAME> to eTUPOOperationStatus='R'
```

2. Submit an LDAP delete request against the account to complete the removal. Using ETAUTIL, the command is as follows:

```
etautil [-d domain] [-u user [-p password]] delete  
'eTUPOAccountContainerName=Accounts, eTUPODirectoryName=<DIRECTORY_NAME>,  
eTNamespaceName=Universal Provisioning ' eTUPOAccount  
eTUPOAccountName=<ACCOUNT_NAME>
```

Note: This procedure does not invoke the DELETE_ACCOUNT exit.

Index

!

!ADD • 115
!bInit conditional clause • 156
!TOUPDATE • 115

.

.dlls
 EtaAttrMap.dll • 23
 etachmap.dll • 23
 EtaConfig.dll • 23
 etaldap.dll • 23
 etalog.dll • 23
 ETAOpenLDAP_Wrappers.dll • 23
 etaptut.dll • 23
 EtaServerMsg.dll • 23
 EtaXML.dll • 23
 eTrustAdminServer.dll • 23
 logapi.dll • 23

A

access this computer from network rights • 115
account containers • 49
account object class • 71
accounts container object • 187
acquisition
 an overview • 79
 correlation • 84
 exploration • 80
 overview of • 22
 registration • 79
 update users • 86
act as part of the operating system right • 115
ADD • 115
 operation • 31
adding a namespace object
 overview of • 62
adding objects
 to the Manager • 61, 206, 207, 208
 using the SASP • 89
administrative directory
 DIT • 31, 39
 overview of • 22
agent DIT

 constructing • 95
agent dlls
 running using the VC++ debugger • 91
agent plug-ins
 building • 51
 code for sample namespace • 65, 210
 enabling parallel search in • 170
 enhancements for program exits • 257
 overview of • 27
 scenario for building • 92
 super agent framework APIs • 289
agents
 building • 171
 coding • 163
 initializing • 167
 installing and testing • 174
APIs
 GUI Framework • 273
 super agent framework • 289
apply() • 273
architecture of eTrust Admin • 19
ASCII • 115
attribute mapping table
 example of • 192
AttributeMap class • 302
attributes
 definition of • 33
 KEYWORD statement • 115

B

back-ends
 defining DITs • 40
 identifying • 39
 overview of • 21
base classes • 69
 bInit • 275
batch utility • 26, 31
 extending • 26
binary PTT files
 viewing contents of • 107
bInit, base class • 275
building
 agent plug-ins • 51
 GUI plug-ins • 50
 parser tables • 105

C

- C++ Agent Dispatcher object • 24, 27
- C++ Compilation errors
 - resolving • 66
- C++ Manager object • 24, 27
- callout module, creating for GUI plug-in • 128
- CanDuplicateEntry • 279
- CAPropertyPage class • 142
- CAPropertyPage.h • 273
- CASE=sensitivity • 115
- Child_Class property • 74, 75
- CLASS statement • 109
 - guidelines for • 103
- classes, deriving • 164
- client interfaces • 26
- code for the sample namespace • 65, 210
- common exits
 - support • 257
- common object classes • 70
- COMPARE operation • 31
- compilation environment
 - refreshing • 66
- compiling the sample namespace • 56, 205
- components of eTrust Admin • 19
- components of the server • 21
- configuring the SASP • 52
- constructing
 - agent DIT • 95
- containers, an overview • 49
- correlation • 64, 84, 209
- COSSchemaAbridged.txt • 263
- COSSchemaUnabridged.txt • 264
- CreateInclusion • 280
- creating
 - virtual attributes • 193
- custom options
 - upgrading • 67

D

- data information trees, see DIT • 31
- DATALOCATION • 40, 103
- DEadd • 289
- DEdelete • 290
- DEFAULT=value • 115
- defining
 - back-end DITs • 40
 - objects in parser tables • 36
 - objects to your schema • 49
 - schemas • 36

- DEinit • 291
- DEinit() function • 96
- DEinitNewObject • 292
- DELETE operation • 31
- DeleteEntry • 282
- deleting nodes • 290
- DEmodify • 293
- DEmodrtn • 293
- DEPENDSON • 115
- DEPRECATED • 115
- DEsearch • 294
- DEsearch() function • 294
- Detach() function • 290
- diagrams
 - object hierarchy • 272
 - UPO Program Exits flow execution • 251
- directory
 - containers • 49
 - object classes • 70
- directory container object • 189
- directory contents
 - in the SDK • 65
- directory schema • 263
- directory structure • 266
- distinguished name, see DN • 34
- distinguished names • 266
- DIT
 - constructing an agent plug-in for • 95
 - declaring object classes in • 94
 - defining back-ends • 40
 - definition of • 31
 - eTrust Admin root domain • 267
 - guidelines for • 100
 - identifying back-ends • 39
 - parent domain • 268
- DMODirectoryEntry.H • 289
- DN • 34, 38
 - renaming • 293
 - searching objects • 294
- DN suffixes • 39
- domain name suffix • 266
- dumpptt • 107
- DuplicateEntry • 283
- DXHOME • 55
- dynamic internal nodes, searching • 294

E

- EDITTYPE
 - list of • 115

- environment, settings • 55
- errorField • 277
- eTAccount base class • 69
- eTAccount object class • 71
- ETACLDAP • 115
- eTAdminProfile object class • 72
- eTAdminProfileContainer object class • 72
- ETAHOME • 55
- ETAHOME directory • 16
- ETASDK • 55
- ETASDKHOME • 55
- EtaServerMsg.dll • 23
- etaupoad • 237
- eTContainer base class • 69
- eTContainer object class • 70
- eTDirectory base class • 69
- eTDirectory object class • 70
- eTExit object class • 72
- eTExitType • 246
 - values for • 246
- eTGlobalGroup object class • 72
- eTGlobalGroupContainer object class • 72
- eTGlobalUser object class • 71
- eTGlobalUserContainer object class • 72
- eTInclusionContainer container • 73
- eTInclusionContainer object class • 73
- eTInclusionObjec • 73
- eTInclusionObject object class • 73
- eTInclusionSubordinate container • 73
- eTInclusionSubordinate object class • 73
- eTInclusionSuperior • 73
- eTInclusionSuperior container • 73
- eTNamespace object class • 70
- eTPolicy base class • 69
- eTPolicy object class • 71
- eTProgramExitContainer object class • 72
- eTRole object class • 71
- eTRoleContainer object class • 71
- eTrust Admin
 - architecture of • 19
 - Server • 21
- eTrust Admin root domain DIT • 267
- eTrust Admin Server
 - DIT • 31, 38
 - overview of • 89
- eTrust Directory • 22
- eTrustAdminServer.dll • 23
- eTUPOAccountStatus • 336
- eTXXXXNamespace • 102

- eTXXXXPolicyContainer • 102
- eTXXXXPolicyContainer object class • 71
- eTXXXXPolicyContainer objects • 101
- execution flow
 - for pre-exits • 255
- exit definitions • 258
- exit invocation requests • 259
- exit references • 258
- exit types • 261
- exploration • 64, 80, 209
- explore and correlate
 - overview of • 64

F

- factory functions, implementing • 166
- file extensions • 107
- format of schema files • 264
- frameworks, in the SDK • 28
- front-end, overview of • 21
- functions, implementing • 165

G

- GetPropertySheetID • 284
- global users
 - object classes • 71
- GUI callout functions
 - CanDuplicateEntry • 279
 - CreateInclusion • 280
 - DeleteEntry • 282
 - DuplicateEntry • 283
 - GetPropertySheetID • 284
 - IsValidDrag • 286
 - OnCommand • 287
 - RenameEntry • 288
- GUI framework
 - overview of • 28
- GUI Framework APIs • 273
 - apply() • 273
 - header file • 273
 - setData() • 275
 - setErrorField() • 277
- GUI objects, defining • 139
- GUI plug-ins
 - adding GUI features • 279
 - building • 50
 - building and linking • 161
 - code for sample namespace • 65, 210
 - creating a callout module • 128
 - creating parser tables • 127

- enhancements for program exits • 257
- implementing • 50
- overview of • 26
- testing • 162

H

header file

- GUI Framework APIs • 273
- Super agent framework APIs • 289

HIDDEN=Boolean • 115

hierarchical namespaces • 75

I

identifying

- back-end DITs • 39
- LDAP directories • 31
- objects • 34

implementing

- agent plug-ins • 51
- GUI plug-ins • 50

inclusion object classes • 73

INCLUSIONCHILD property • 75

INCLUSIONPARENT property • 75

inclusions

- common object classes • 70
- examples of • 74
- object classes • 73
- objects • 73
- optional • 75
- optional inclusion • 78
- predefined • 74
- required • 74
- required inclusion • 77
- sample code • 77

INCREMENTAL • 115

initializing agents • 167

initializing objects • 291, 292

Input XML Buffer (Input Argument) • 238

interpreting program exit results • 260

invoking program exits • 260

IsValidDrag • 286

J

Java IAM SDK

- installing • 329
- overview • 329
- sample programs • 330
- SDKSample.properties • 330
- setting up • 330

JIAMJAR

- explanation of • 57

K

KMS namespace

- creating • 198

L

LDAP

- ADD • 289
- DELETE • 290
- directory • 22
- identifying directories • 31
- MODIFY • 293
- MODRDN • 293
- performing operations • 31
- routing requests • 40
- SEARCH • 294

LDAP object names

- common objects • 267

leaf nodes

- deleting • 290
- searching • 294

leaf objects • 49

libodbc++ • 202, 204

libodbc++ library

- downloading and building • 204
- LIBODBCXX_PATH environment variable • 204

logging component • 291

M

Manager • 26

- adding objects • 61, 206, 207, 208

mappings

- global user • 182
- global user to namespace • 183

MAXLEN=integer • 115

MAXVALUE=integer • 115

menu commands • 287

MFC controls • 273

MINABBREV=integer • 115

MINLEN=integer • 115

MINVALUE=integer • 115

MODIFY operation • 31

MODRDN operation • 31

multi-valued attributes • 33

mult-valued attributes • 75

N

- NAMEPROPERTY=ldap_property_name • 109
- namespace
 - containers • 49
 - naming conventions for • 101
 - object classes • 70, 109
- namespace container object • 191
- NAMESPACE namespace_name • 108
- namespace servers • 27
- NAMESPACE statement • 108
- NAMESPACE variable • 57
- native exits • 257
- NMAKE
 - running • 205
 - running • 56
- nodes
 - deleting • 290
- non-hierarchical namespaces • 75
- NOTALLOWSPACE=boolean • 115

O

- object classes
 - in DITs • 94
 - naming conventions for • 101
 - overview of • 32
 - predefined • 69
- object hierarchy • 272
- object processing, using the SASP • 89
- objects
 - accounts container • 187
 - adding GUI features • 279
 - adding using the SASP • 89
 - attributes • 33
 - classifying • 32
 - creating inclusions • 280
 - definition of • 31
 - deleting • 282
 - directory container • 189
 - displaying property sheet • 284
 - drag over another object • 286
 - duplicating • 279, 283
 - identifying • 34
 - initializing • 291, 292
 - menu commands • 287
 - namespace container • 191
 - searching • 294
 - user-friendly names for • 270
- objects and attributes
 - naming conventions for • 101

- OBSCURED • 115
- ODBC SDKAccount_Property.pti • 184
- ODBCWrapper
 - overview of classes • 301
- OdbcWrapper Class • 200
- OnCommand • 287
- OVERRIDE • 115

P

- pAddOp • 292
- parallel search
 - enabling in agent plug-ins • 170
- parent domain DIT • 268
- Parent_Class property • 74, 75
- parser table • 184
 - enhancement • 257
- parser tables
 - attribute keywords • 115
 - CLASS statement • 109
 - creating • 36, 105
 - creating for GUI plug-in • 127
 - defining objects • 36
 - definition of • 35
 - enhancements • 257
 - NAMESPACE statement • 108
 - naming • 107
 - overview of • 26
 - SCHEMAGEN • 37
- pDeletOp • 290
- policy object classes • 71
- POLICYSYNC • 115
- populate
 - code for sample namespace • 210
- populate command
 - update • 197
- post exits • 256
- post-installation steps • 16
- pProposedParent • 292
- predefined
 - inclusions • 74
 - object classes • 69
- pre-exits • 255
- priority of post exits • 256
- priority of pre-exits • 256
- process flows
 - post exits • 256
- program exits
 - code examples • 262
 - interpreting results • 260

- invoking • 260
- object classes • 72
- requests • 259
- types • 261
- Property page
 - adding GUI features • 273
 - initializing • 275
 - setting error flag • 277
 - updating attributes • 273
- property sheet
 - creating code for • 142

R

- RDN • 34
- RDTutility.bat
 - calling directly • 58
 - syntax for • 59
- refreshing
 - compilation environment • 66
- registering • 63, 209
- registering the SDK directory
 - overview of • 63
- registerLogComponent() • 291
- registration • 79
- relative distinguished name, see RDN • 34
- RenameEntry • 288
- reporting error messages • 115
- required objects
 - creating • 101
- return XML buffer
 - for UPO program exits • 245
- rights
 - access this computer from network • 115
 - act as part of the operating system • 115
- root DN • 34
- routing LDAP requests • 40

S

- sample namespace
 - adding objects to the Explorer • 61, 206, 207, 208
 - building steps • 56, 203
 - code • 65, 210
 - compiling steps • 56, 205
 - discovering a directory • 63, 209
 - exploring accounts • 64, 209
- SASP
 - configuring • 52
 - object processing • 89

- overview of • 24, 89
- using • 90
- schema
 - defining • 36
 - defining objects • 49
 - description • 263
 - for directories • 263
 - format • 264
- SCHEMAGEN • 37
 - output location • 37
 - utility location • 37
- SCOPE_BASE search • 294
- SCOPE_ONELEVEL search • 294
- SDK
 - Developer Guide
- overview • 15
 - directory contents • 65
 - framework overview of • 28
 - post-installation steps • 16
 - sample namespace • 56, 203
- downloading and building the libodbc++ library • 204
- SDKAccount_Property.pti • 198
- SDKHOME directory • 16
- search
 - dynamic internal nodes • 294
 - leaf nodes • 294
 - objects • 294
 - static internal nodes • 294
 - types • 294
- SEARCH operation • 31
- server
 - components of • 21
- server configuration
 - OpenLDAP (SLAPD) • 22
- server framework • 23, 89
 - base classes • 69
 - overview of • 28
- server plug-ins
 - overview of • 89
- server plug-ins, overview of • 23
- setData() • 156, 275
- setErrorField() • 277
- setSdsData() routine • 273
- setting your environment • 55
- SHEET=boolean • 109
- single-valued attributes • 33
- SLAPD
 - overview of • 21

- reading DNs • 40
- SMPLUGINDLL=SASPRgiester • 102
- static internal nodes, searching • 294
- suffixes • 39
- super agent
 - configuring • 52
 - DIT • 31, 39
 - framework • 29
 - framework APIs • 289
 - multi-valued attributes • 75
 - overview of • 23
- super agent framework APIs
 - DEadd • 289
 - DEdelete • 290
 - DEinit • 291
 - DEinitNewObject • 292
 - DEmodify • 293
 - DEmodrdrn • 293
 - DEsearch • 294
 - header file • 289
- SUPERCLASS property • 69
- support for common exits • 257
- SyncRemoveValues • 115

T

- testing
 - GUI plug-ins • 162
- TOUPDATE • 115

U

- update users • 86
- upgrading
 - custom options • 67
- UPO Program Exits
 - architectural overview of • 236
 - authentication type • 243
 - code examples • 252
 - etaupoad • 237
 - implementing in multiple domain environments • 237
 - order of • 237
 - overview of • 235
 - return XML buffer • 245
 - sample flow execution diagram • 251
 - structure of • 238
 - XML Account block • 241
 - XML Exit Custom Data block • 244
- user-friendly names
 - for objects • 270

- user_friendly_name • 264
- UTF8 data • 115

V

- VALUE=value • 115
- VC++ debugger
 - using to run agent dlls • 91
- VERBREQ=limitation • 115
- virtual attributes
 - creating • 193
 - example • 194
- VSVARS32.BAT • 56, 205

W

- WIN32DLL=dll_name • 109
- WIN32HELP=help_file • 109

X

- XML Account block • 241
- XML Exit Custom Data block • 244
- XML Operation block • 239